University of Oslo
Department of Informatics

A high performance cluster file system using SCI

John Enok Vollestad

Cand Scient Thesis

Autumn 2002

Abstract

This thesis is about interaction between different architectures in high performance computing for file system I/O. This is evaluated by performance, scalability and fault handling. What excel in a loosely coupled system fail in a tightly connected system and vice versa.

The I/O-path from disk to application have been examined both theoretically and with tests for local and distributed file systems. The impact of different levels of cache is shown using various tests.

This test results has been used to design and implement a protocol giving Scalable Coherent Interface (SCI) the semantics of TCP/IP, thereby replacing TCP/IP in Parallel Virtual File System (PVFS). SCI is a low latency, high throughput interconnect with decentralized routing. In PVFS interconnect latency have only proven important for meta data operations. For I/O operations the pipelining hides the latency with the protocol window. PVFS have as expected shown increased read and write performance with increased interconnect throughput. Throughput have been increased by a factor of 5 by introducing SCI from 100Mb/s Ethernet. To limit overloading in the interconnect, two different techniques have been evaluated. Exponential backoff as in TCP/IP and a token based scheme.

Preface

Due to job, marriage and some other factors this thesis was written over a period of three years. During that time it has changed considerably as the focus of the thesis and my competence have changed.

Acknowledgments

I would like to thank my tutor Knut Omang for guidance and putting up with me when this thesis was not my first priority. Since Knut changed jobs during my thesis I have in the later part of my work had to rely on Terje Eggestad and Lars Paul Huse for answering questions and giving guidance in technical parts. Håkon Bugge shed light on some of the problems which I am grateful for. Erik Vasaasen helped with the language since English is not my native.

I would also give a big thanks to Scali for letting me use machines and offices space whole testing the SCI implementation of PVFS.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The focus for this thesis is clusters for computational and I/O intensive applications using Message Passing Interface (MPI) and SCI for high performance computing. Currently shared data is either transferred using MPI between applications when needed or placed on Network File System (NFS) or PVFS.

A distributed file system creates a mechanism for sharing large data sets while running I/O intensive distributed applications. These distributed applications are often very parallelized, dividing load among more hardware leading to increase performance. It is thus important for the file system to support this as well. Such clusters of computers might have more than 100 individual computer nodes and $200 - 300$ client applications. The file system should therefore support scaling to more nodes without introducing inconsistency.

The qualities needed by a shared file system are:

- High read/write performance with many simultaneous clients both on different and same data sets.
- Strict consistency.
- Ability to scale, both in data size and number of simultaneous clients.

On the above list NFS falls short both with performance for many simultaneous clients and ability to scale. Strict consistency can be achieved only with loss of performance.

In the rest of this thesis I document possible ways to achieve the above qualities by doing surveys, evaluate and compare solutions for a generic storage system, and doing comparisons with standard I/O-systems in order to reveal the pros and cons of the various products.

1.1 Background

To implement a distributed file system that provide the qualities listed above it is important to understand the environment it is going to operate under.

Cluster technology was introduced because there are tasks that are too demanding in cpu-time or data-size to be solved on a single high-end workstation

in a reasonable time. There are two main approaches for solving this kind of problems:

One way is to construct a more powerful computer. The downside of this is that the computer is made especially for a task or a limited range of tasks and consumers. The computer is therefore made in a small number making it more expensive than a standard computer to the degree that the cost/performance ratio is lower.

The other way to solve the task is to divide the problem and let each part be solved on a standard high-end workstation. Now the performance cost ratio is better but there are still obstacles. Often it is difficult to divide the problem into suitable pieces for the individual workstations, and when done there is a need to synchronize the results from each workstation. In addition data has to be exchanged between workstations during the task. Sometimes the amount of data is large, making the bandwidth and latency of the network interconnects very important.

1.1.1 Different file systems

The I/O demands of traditional applications have increased over time as shown in "Measurements of a distributed file system" [BHK+91]. Multimedia, process migration and parallel processing increase file system demands. Just a few workstations simultaneous running video applications would swamp a traditional central server [Ras94].

1.1.1.1 Storage systems generally

A storage system might be used both passively as long term storage and actively as a communication medium. The last may seem strange, to use slow disk when a direct network connection between individual workstations should be able to do this much faster, but sometimes disk is still the better choice:

- *Size* If the data to be transferred between processes is too large to fit in memory.
- *Time* If the time between each accesses is long.
- *Interrupt the recipient* If the recipients CPU cache becomes partially broken when it receives the data. One might argue that network buffers might be excluded from caching when receiving data not requested, but I do not know of any such implementation.
- *Not all data* If not all the data is interesting there is no reason to waste RAM.
- *ACID* Some systems use some or all the Atomicity, Concurrency, Isolation and Durability (ACID) properties. This eases programming.

Another advantage with shared media is the possibility of taking advantage of specialized hardware in centralized servers, thereby gaining the opportunity for centralized administration and management. The data may be logically connected and should therefore be stored in the physical same place.

1.1.2 I/O path

Data to and from a file system has to travel a path called the I/O path. Bottlenecks in this path decrease performance and are therefore important to understand. Chapter 2 shows how data travels from application to disk, and what can be done to increase throughput.

Based on this, several distributed file systems have been evaluated for the high performance cluster environment described at the start of this chapter. Using the knowledge from chapter 2 a protocol for use over SCI with features resembling TCP/IP have been made and used in PVFS instead of Ethernet to increase performance and for testing the impact of latency. PVFS is further described in section 3.1.5.

1.2 The structure of the thesis

Clustering for high performance is a large field with lots of literature. This have mostly been new for me which meant that I had to do a survey on the subject of cluster file systems. This is part of my contribution. The knowledge gained from the survey have been used to select an existing file system and to extend this to use SCI as the network interconnect. This changes accumulated to about 3200 lines of code including a library for simplified communication with TCP/IP attributes using SCI and a plugin-based adapt layer for simultaneous use of several protocols.

The test results match the theory in the survey but at the time some were not as expected. The smaller latency of SCI were to believed to have larger impact than it had. This led in turn to some of the survey.

Chapter 1 gives an introduction to the thesis by briefly touching the different subjects visited.

Chapter 2 shows the I/O path with bottlenecks and how it all comes together.

Chapter 3 gives an overview of existing file systems and how they cope as a cluster file system.

Chapter 4 Gives an overview of my work in making PVFS use SCI as the interconnect.

Chapter 5 summarizes the results of this thesis.

Appendix A contains a vocabulary for most of the acronyms and expressions used in this thesis.

Chapter 2

Local I/O performance

5-10 years ago the local I/O throughput was much higher than that of fast networks at the time. Local I/O throughput had a low impact when transferring data over a network. Now it is beginning to become an issue. The bandwidth of a single optical fiber has been doubling every 16 month since 1975 [Gon01] but the bandwidth of the internal bus has been doubling only about every 3 years[Inf00]. Table 2.1 show the performance levels as they are today. Note that the Peripheral Component Interconnect (PCI) performance will vary with different chipsets.

The latest SCI hardware have a throughput of 326MB/s, which is higher than most PCI chipsets. Because of this the local I/O performance is very important for a distributed file system when using high throughput networks such as SCI. There seems to be a crossing point where the network creeps into the machine and replaces the bus [Inf00]. If that should happen low latency network is a necessity.

For comparison both local and distributed file systems are outlined in figure 2.1. In most cases the network have lower throughput than that of the rest of the I/O pipeline, with the possible exception of disk if it is not striped. Even if the network throughput is infinite, the throughput of the PCI bus is $\frac{1}{5}$ of that of memory. A distributed file system can never be faster than a local

	Throughput	Latency
CPU cache	3200 - MB/s	-
Memory	3200MB/s	6 - 12ns
SCI interconnect	70 - 326MB/s	7 - 1.4us
Myrinet	245MB/s	7us
1GB Ethernet	112MB/s	-
100Mb Ethernet	12.5MB/s	90us
10Mb Ethernet	1.25MB/s	90us
PCI 64bit, 66MHz	486MB/s	-
PCI 64bit, 33MHz	235MB/s	-
PCI 32bit, 33MHz	121MB/s	-
SCSI Ultra 320	320MB/s	-
Ultra ATA 100	100MB/s	-
Disk	15 - 50MB/s	9ms

Table 2.1: I/O max sustained performances

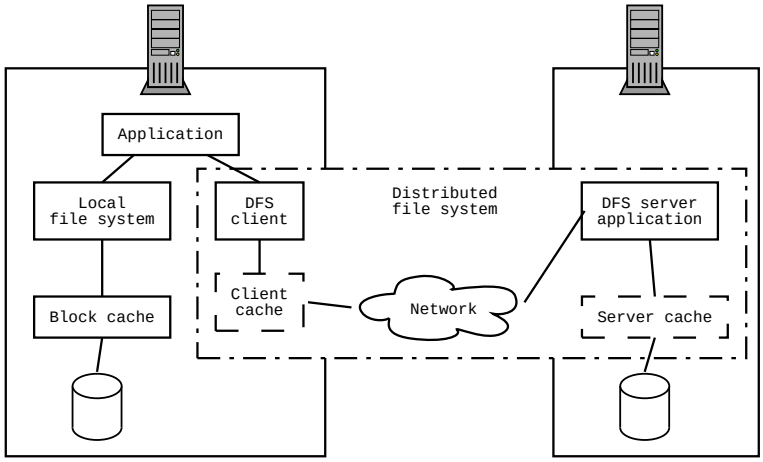

Figure 2.1: Local and distributed file systems

file system can be, even when using a high throughput network. Adding more components for the data to travel through can not improve performance.

With increased network throughput as with SCI in table 2.1 the network is no longer a bottleneck and other parts of the system become bottlenecks. The best SCI cards were not available for this thesis so to be able to evaluate what a system with higher network throughput would be limited of, tests were done on local I/O.

Local file system contain parts that is present in a distributed file system and without the network in a local file system as with increased network throughput the network is no longer a bottleneck and other bottlenecks become more apparent. Study of local file systems is therefore useful to understand the impact of increased network throughput.

This chapter shows impact on the local I/O path depending on access pattern, record size and data size. The tests shown here have been done to reveal if there is substantial differences in performance within and between two computers with different architectures, a Sparc server and an Intel x86 PC respectively. There are huge differences between the two in how the performance changes depending on different access patterns.

2.1 Bottlenecks

In file systems performance is characterized by throughput and response time (latency). It is difficult to get both high throughput and low latency at the same time. High throughput demands high utilization of the data-pipeline. This means that operations in the pipeline have to be optimized to run in parallel by using buffers. Low latency need fast handling and therefore least possible amount of code to be run and buffers to be copied between. This can be reduced by reduce the amount of copying and thereby latency as in “Incorporating Memory Management into User-Level Network”[WBvE97].

Amdahl's law:

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

Obviously, reducing the already short time intervals gives less overall speedup than for longer ones. This chapter therefore focus on the largest bottlenecks first. This is important because optimizing one part may place restrictions on what optimizations can be done elsewhere and the biggest bottleneck have the biggest possible optimization gain and should therefore have the first and best choice of optimizations.

2.2 PC versus server

It is no longer unusual to use ordinary PCs as servers. Earlier it was a widespread opinion that servers should be specially built to be servers.

One of the test machines is an ordinary PC with a Pentium III processor. The other is a Sun Ultra 1 with a Sparc processor. The later is a computer built as a dedicated server. The different targets for the two computers has its results in the tests. The PC is a lot newer than the Ultra 1 and has a higher peek performance, but it is also slower in some situations. Note that only throughput have been tested, not speed of updating metadata on local file systems.

Computer one PC with Intel architecture running an Pentium III at 600 MHz using the Linux 2.2.16-3 kernel using a the Second Extended File system (Ext2) file system on an EIDE disk. It has 128 MB RAM, 16 KB L1 cache for instructions and an equal size cache for data, and 256 KB L2 cache on the CPU. This is a somewhat old kernel and I/O performance have increased with newer kernels but the tests done here is mostly for the relative difference in performance and not for the exact numbers. The Pentium III chip was introduced in the middle of 2000.

Computer two Sun Ultra 1 with a Sparc architecture running at 167MHz on SunOS 5.7 using an UFS file system on a SCSI disk. This file system was used for both /tmp and swap but the system were not swapping at any time during the test. The computer has 128 MB RAM, 16 KB L1 cache for instructions and equal size for data, 512 KB L2 cache. This computer is no longer in production and is used here because of the architecture being targeted at servers. The Ultra 1 was introduced in late 1995.

2.2.1 Architecture

More users increase randomness in the total ordering of data accesses. More users also increase the amount of data handled simultaneously on the server. Random operations on large data sets are therefore appropriate for testing a systems performance as a server.

Test results for random writes on the Linux and the Sun OS system are given in figure 2.2 and 2.3. The PC have very high performance when the amount of

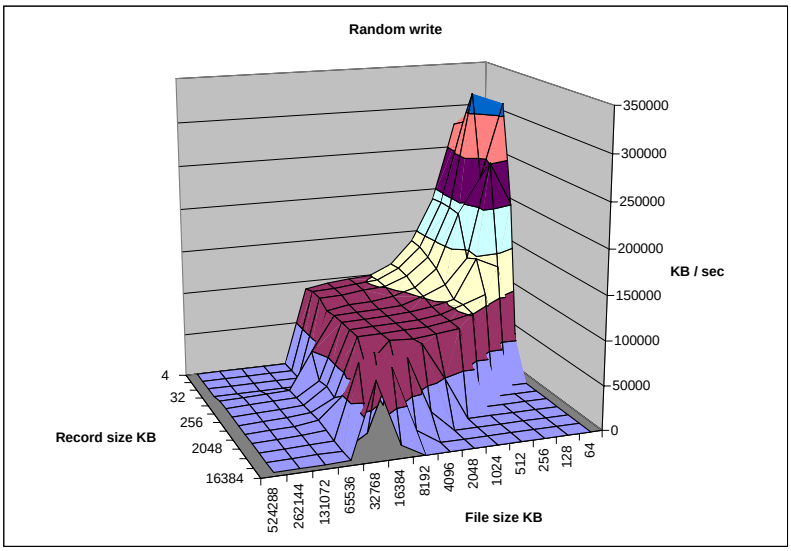

Figure 2.2: Linux random write

written data is small but only one third the performance for files twice the size of the L2 cache. The server on the other hand show little change in performance for file size. The operating systems of each computer also contribute a great deal to the performance. The Ultra 1 have disabled CPU cache for memory copy operations.

The drop in performance for record sizes less than 32KB for files larger than 32MB is incorrect. The test did not use record sizes less than 32KB for files of 32MB or larger to save time.

For random read the results are more equal for large record sizes. Test results are shown in figure 2.4 and 2.5. For small record sizes the PC perform 3× the performance for large record sizes. A possible reason for this is given in section 2.3.

For continuous operations the PC show higher performance than the server. Graphs of these tests are given throughout the rest of this chapter where they are part of explanations of local I/O.

The main difference between the Linux PC and the SunOS server is that the performance drops very much with the file sizes on the Linux PC and nearly not at all on the SunOS server. That is, the SunOS box has a drop in throughput, but that is at file sizes larger than the memory buffer size. The Linux box performance is more uneven for various sizes.

Both the computers have 128MB of RAM and the test file size does not go higher than 512MB, but this is sufficient as at 256MB it is larger than the main memory. The raw throughput is therefore successfully tested. The test results from file sizes less than 128MB are also interesting because in most situations almost all files on a file system will be smaller than this size. The results also

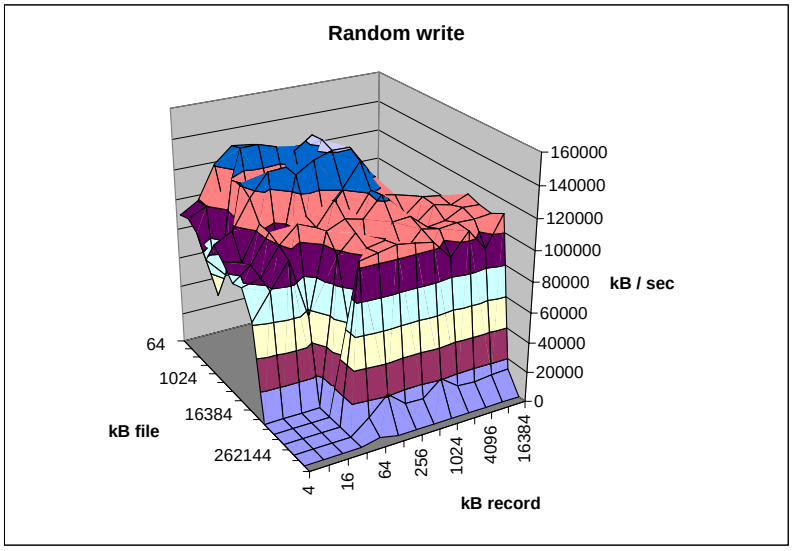

Figure 2.3: Sun random write

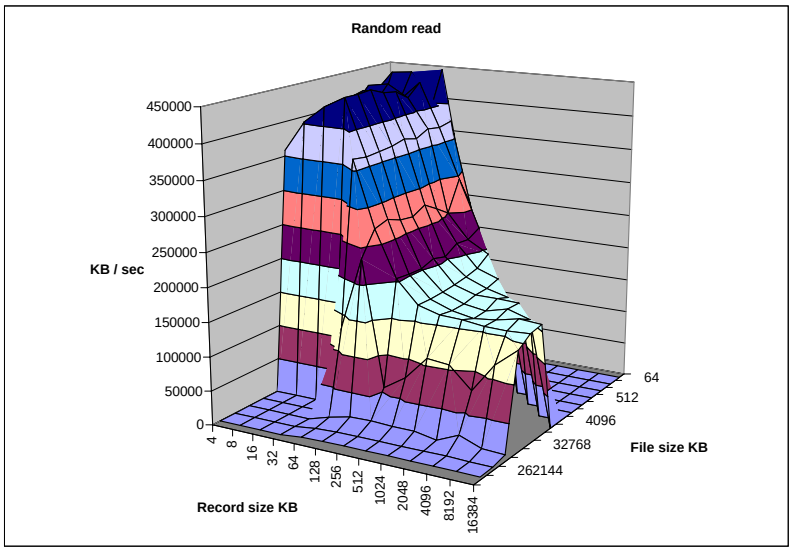

Figure 2.4: Linux random read

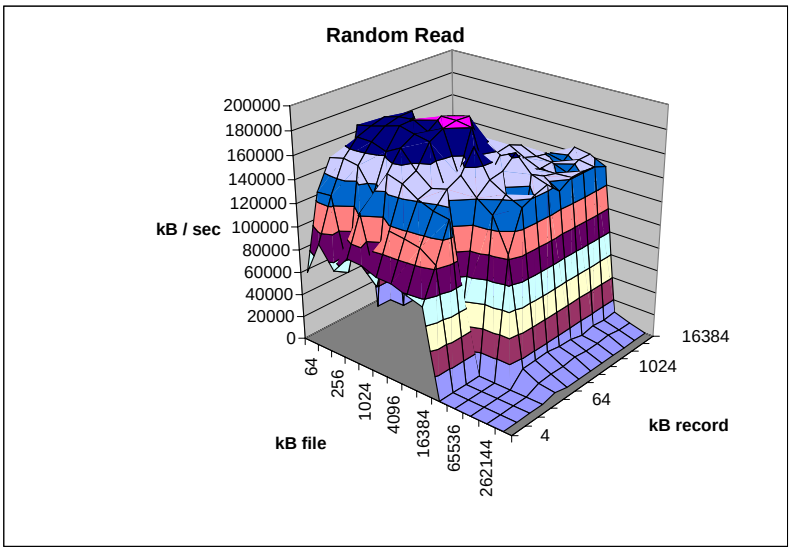

Figure 2.5: Sun random read

apply if only a part of a larger file is used at a time. That is, if the part is less than the buffer size.

2.2.1.1 Test results from the Linux box

Since the test results on this machine showed the most variation, I've used them to illustrate the different aspects of the I/O pipeline.

The machine has good peek performance. As it is much newer than the Ultra 1, this is no surprise. It has however much more variation in performance over file and record sizes. The tradeoff here is that it performs great as a desktop computer and less well as a server with a high load. It should be mentioned that more memory would increase the size of the disk cache hit rate and therefore improve the performance when working on large data sets.

The EIDE bus has a lower performance than the SCSI bus in the Ultra but this should not create the big drops in performance for the file sizes larger than 8-32MB dependent on test. As there were other processes running on the computer as the test were run it might appear that not enough memory was available for disk buffering during the test.

2.2.1.2 Test results from the Ultra 1

It seems clear that the machine has a good I/O performance. The small changes in performance for files sizes under 64MB makes it a better server since more clients increase the amount of data handled simultaneously.

The main memory that is not used for other purposes is used as a buffer cache. At the time the test was run there were no other large processes. The

performance fall at 64MB and larger file sizes is due to these files not fitting the buffer cache .

The local partition used on the computer was not big enough to run the whole test. The data is therefore not reliable on corresponding values to file sizes at 512MB. This should not matter as the main memory available for buffering was less than 128MB, making the test results at 256MB file sizes sufficient.

2.2.1.3 Comments

There is huge differences in how a computer access local data and how a computer serves multiple clients. For the local data the CPU cache provide a great boost. For a server with many clients there might be better to disable the CPU cache for data and instead use it for instructions. With many clients the randomization increases and the possibility for hit in the CPU cache decreases.

2.3 Data pipeline

A CPU waiting for I/O is not utilized. When utilization drop, efficiency and performance drop. To ensure that CPUs wait as little as possible a pipeline is implemented.

Pipelining is a technique where multiple operations are overlapped in execution. This increase the overall execution speed. When accessing a file system on a local or remote computer the data travels through a data pipeline where several operations might run in parallel in different components. A components is here a hardware part with its own cpu and some memory, note that this apply to network cards and disk controllers in addition to the main CPU. The essential concept behind pipelines is to use buffers to hold the data between components.

2.3.1 Buffers

A buffer helps with adaptation in communication between components that send and receive with different speeds and record sizes. It also makes it possible for the components to send or receive data when suitable, thereby limiting complexity and increasing efficiency. Equation 2.1 shows simplified the minimum buffer size to avoid performance drop between two components sending with different bandwidth and periods of no communication.

$$S = MAX(R_{buffer},\ \ S_{buffer}) \tag{2.1}$$

Where: S is the minimum size of the buffer to avoid performance drop.
$R_{buffer} = S_{\Delta T} \times R_{bandwidth}$
$S_{buffer} = R_{\Delta T} \times S_{bandwidth}$
$R_{\Delta T}$ is the maximum time between receiving.
$S_{\Delta T}$ is the maximum time between sending.
$R_{bandwidth} < S_{bandwidth}$

With higher bandwidth buffers are filled faster and buffers generally need to be larger. The standard network buffer size in the Linux 2.4 kernel is 64KB. This can, and should, be increased when using high speed networks. The SCI protocol

implemented and presented in chapter 4 showed an increase in performance using buffer sizes as large as 1MB when used in collaboration with PVFS.

The overall throughput in a data pipeline can be no better than the lowest throughput for any component in the pipeline (see equation 2.2). This makes it is possible to increase the performance of an individual component of a system without increasing the performance of the total system. *Bottlenecks* are the components where an increase in performance improve the total performance.

$$O_t = MIN_{i=n}^{m}(C_i) \tag{2.2}$$

Where: O_t is the overall throughput.
C_i is the throughput of component i in the data pipeline.
n to m are identifiers for each component in the pipeline.

2.3.2 Record size

The record size influence the throughput of pipeline. If the record size is much smaller than the smallest buffer in the pipeline then the components in the pipeline will use an almost ignorable extra time to handle more records. If on the other hand the record size is larger than the smallest buffer in the pipeline, overlapping execution is stopped as the components have to wait for the next component in the pipeline to start transmitting the data before sending the next record. This delay will then propagate through the pipe and slow the whole pipe down.

It is important to remember that when an application sends data it is the component at the start of the pipeline and the same rules apply to it as the rest of the parts in the pipeline. If the record size of the application is larger than the smallest buffer in the rest of the pipeline, performance drops. This is clearly visible in figure 2.4 and 2.6 where smaller record sizes up to 32KB show about 40% higher throughput.

2.3.3 Performance

Table 2.1 shows throughput and latency for different parts of the pipeline. It is important to note that these values sustained throughput but on the best hardware. 15% difference in throughput for PCI chipsets of same width and frequency have been observed. It is expected that similar differences exists for the others. Above it is explained why a small buffer size will decrease performance. In the rest of this chapter other reasons to performance drop from this numbers will be stated.

Figure 2.1 shows the data pipeline for both local and distributed file systems. The local block cache and the client cache are placed in memory and have an equal throughput to that of local memory. Throughput from the memory cache on a remote computer is equal to the network throughput or the PCI bus, whichever has the lowest throughput in the data pipeline. This means that throughput from memory cache on a remote computer over a network at least 5× slower than local/client memory cache.

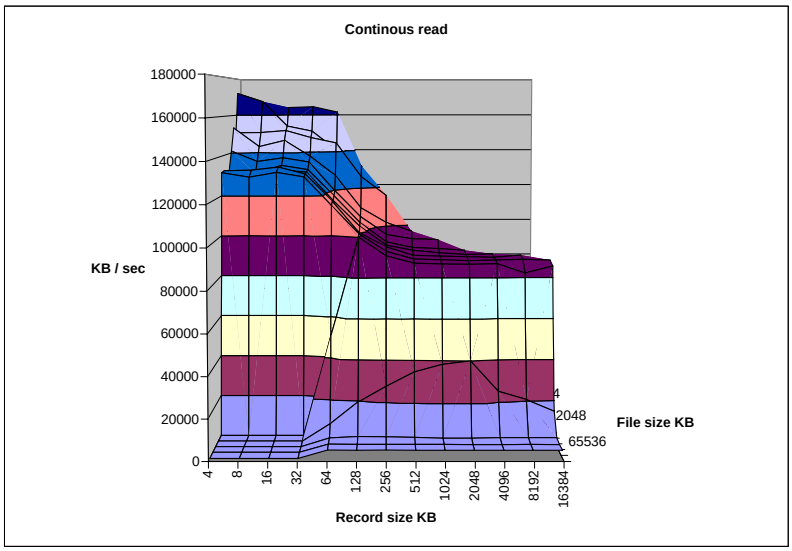

Figure 2.6: Linux continuous read by record size

2.4 Disk

Disk access and throughput is the slowest component in a local file system. The numbers in table 2.1 show that it has a throughput of $< \frac{1}{50}$ of memory. In this section it is shown how to limit this bottleneck.

IBM invented hard disk drives in 1957. In 2002 capacity increases logarithmic per year and price falls logarithmicallyáaccording to IBM. Access times are greatly reduced for the accesses that exploit the on-board disk caches but access times to the disk plates themselves have not improved accordingly and remain a major bottleneck in disks today.

There are three major approaches for improving the bandwidth of disks:

- *RAID* as in "A Case of Redundant Arrays of Inexpensive Disks" [DAP88]
- *Caching* as in "Caching in the Sprite Network File System" [NWO88]
- *Log-structuring* as in "The design and implementation of a log-structured file system" [RO91].

To improve access time only caching shows results that have proved really usefull in generic systems.

2.4.1 RAID

RAID [DAP88] stand for Redundant Array of Inexpensive Disks but because of the restrictiveness of Inexpensive, sometimes RAID is said to stand for Redundant Arrays of Independent Disks. It makes a single volume out of several

Type	Performance	Fault tolerance
RAID-0	Highest throughput of all RAID types	$MTBF = \frac{each\ disk\ MTBF}{number\ of\ disks}$
RAID-1	-	Mirroring, possible to reconstruct defect disk
RAID-2	Possibly high	"On the fly" data error correction
RAID-3	High if implemented in hardware	May tolerate disk failure, reconstructing failed disk is possible
RAID-4	High on read, low on write	May tolerate disk failure, difficult and inefficient data rebuild
RAID-5	High if implemented in hardware	May tolerate disk failure, difficult data rebuild
RAID-6	Very poor write performance if not implemented in hardware, possibly better performance	Very good, can handle failure of two disks
RAID-7	Overall write performance is 25% to 90% better than single spindle performance and 1.5 to 6 times better than other array levels	-
RAID-10	Potential throughput as RAID 0 if support in hardware	As with RAID 1
RAID-53	Higher than RAID 3	Good

Table 2.2: RAID levels

disks and thus improves throughput and/or fault tolerance. Higher throughput is achieved by reading and writing in parallel to multiple disks. Fault tolerance is done by adding redundant data on other disks. If a disk fail, this redundant data can be used to reconstruct the lost data on a new disk later or even on the fly. The different raid levels combine these two techniques with different emphasis to obtain the wanted qualities.

It is important to notice that RAID does *not* improve the access time of the volume, as all disks are read in parallel. The individual disk properties remain unchanged.

Spindle synchronization means that the rotation of the platters is synchronized. This was used more widely earlier for high speed throughput. Now the rotation speed have increased and RAID hardware compensate of unsynchronized spindles. Therefore synchronization is less used.

2.4.1.1 RAID levels

Table 2.2 gives an overview of the different RAID levels. RAID level 2, 7 and 53 are today less used because of the specialized hardware for spindle synchronization.

Raid-0 is striped disk array without fault tolerance Raid-0 provide non-redundant striped drives without parity. Because it is non-redundant, if any drive crashes, the entire array crashes. It offers highly efficient RAID data storage, but lowest level of security. (Theoretical 2x write and 2x read speed)

Raid-1 is mirroring and duplexing The same data is written to pairs of drives. It has therefore twice the write transaction rate of single disks and same

read transaction rate as single disks. If one drive fails, its matching pair may be used (if implemented such.) Best performance of any redundant RAID array and highest cost of redundancy.

Raid-2 uses hamming code ECC "On the fly" data error correction and extremely high data transfer rates possible. It is inefficient because of the very high ratio of ECC disks to data disks with smaller word sizes. Transaction rate for write is equal to that of a single disk at best (with spindle synchronization).

Raid-3 is parallel transfer with parity Data is striped across 3 or more drives. Example (using 3 drives): Half the data is stored on drive 1. Half the data is stored on drive 2. Parity information is stored on drive 3. If any drive fails, then its data can be recreated using the other 2 drives. Very resource intensive to do as a software RAID.

Raid-4 is independent data disks with shared parity disk It has very high read data transaction rate and worst write transaction rate and write aggregate transfer rate. Low ratio of ECC (Parity) disks to data disks means high storage efficiency. Difficult and inefficient data rebuild in the event of disk failure.

Raid-5 is independent data disks with distributed parity blocks RAID-5 stripes information across disks, storing parity information as per RAID-3. In RAID-5, no single drive is reserved as the parity drive. Parity and data is striped on all drives. Because no single drive is the parity drive, bottlenecks are avoided. Most complex controller design. Difficult to rebuild in the event of a disk failure (when compared to RAID level 1).

Raid-6 is independent data disks with two independent distributed parity schemes It is essentially an extension of RAID level 5 which allows for additional fault tolerance by using a second independent distributed parity scheme. Provides for an extremely high data fault tolerance and can sustain multiple simultaneous drive failures. Controller overhead to compute parity addresses is extremely high. Very poor write performance.

Raid-7 is optimized asynchrony for high I/O rates as well as high data transfer rates This is a single vendor proprietary solution. Extremely high cost per MB. Overall write performance is 25% to 90% better than single spindle performance and 1.5 to 6 times better than other array levels. Small reads in multi user environment have very high cache hit rate resulting in near zero access times.

Raid-10 very high reliability combined with high performance Is implemented as a striped array whose segments are RAID 1 arrays. Very expensive. Excellent solution for sites who would have otherwise gone with RAID 1 but need some additional performance boost.

Raid-53 High I/O Rates and Data Transfer Performance RAID 53 Should really be called "RAID 03" because it's implemented as a striped (RAID level 0) array whose segments are RAID 3 arrays. Very expensive to implement. All disk spindles must be synchronized, which limits the choice of drives. Maybe a good solution for sites who would have otherwise gone with RAID 3 but need some additional performance boost.

2.4.1.2 RAID on RAID

For the cause of compatibility the disks in a RAID is seen as a ordinary disk by the hardware and software connecting to it. This means that there is possible to connect RAID devices as disks on other RAID devices. Mostly this is done with the combination of software or internal on one RAID controller. This since hardware RAID is most often realized as disk controllers and therefore not possible to connect to other RAID hardware. Examples of this is to run RAID-1 on RAID-0 devices. Because there is no parity calculation involved the throughput can be as high as RAID-0 with the fault tolerance as RAID-1. This RAID on RAID setup is often referred as RAID-10 or RAID-0+1. Many controllers support this directly.

2.4.1.3 Performance potential

Table 2.1 show that disk have $< \frac{1}{50}$ of the throughput of memory. Using RAID-0 and 10 disks reduce this to $< \frac{1}{5}$. If any redundancy is applied as in raid 4 or 5, hardware support is needed to avoid using memory bandwidth for calculation of redundancy data. Maintenance of RAID parity also strain write performance because of the read-modify-write sequence it uses.

2.4.2 Memory as disk cache

Memory throughput is about 50× faster compared to disk. Memory access time is $\frac{1}{10^6}$ of the disk access time. Conventional memory used as a disk cache avoids some of the slow disk operations and increase performance considerably. Figure 2.7 and 2.8 show the impact of cache. Raw disk throughput is barely visible as $4 - 5$MB/s in the front and left lower part respectively.

$$H = \begin{cases} C < D & \Rightarrow & \frac{\mathrm{C}}{\mathrm{D}} \\ C > D & \Rightarrow & 1 \end{cases} \tag{2.3}$$

Where: H is the hit rate with random access patterns and a warm cache
C is the cache size
D is the data set size

It is important to note that the part of the data set that is subject for change is mostly much smaller than the complete data set. It is also to a certain degree possible to predict the next area subject to read before it is issued. This is called prefetching and the most efficient and best known is to continue to read the next bytes from where the last stopped. It works because most operations are continuous.

Read cache When a source of data is slower than the consumer of the data and the same data is repeatedly requested, a buffer of earlier read data

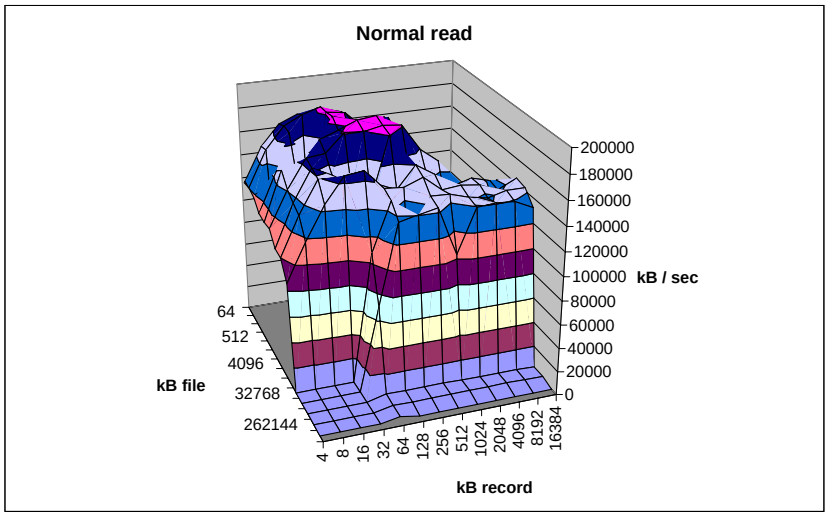

Figure 2.7: SunOS continuous read

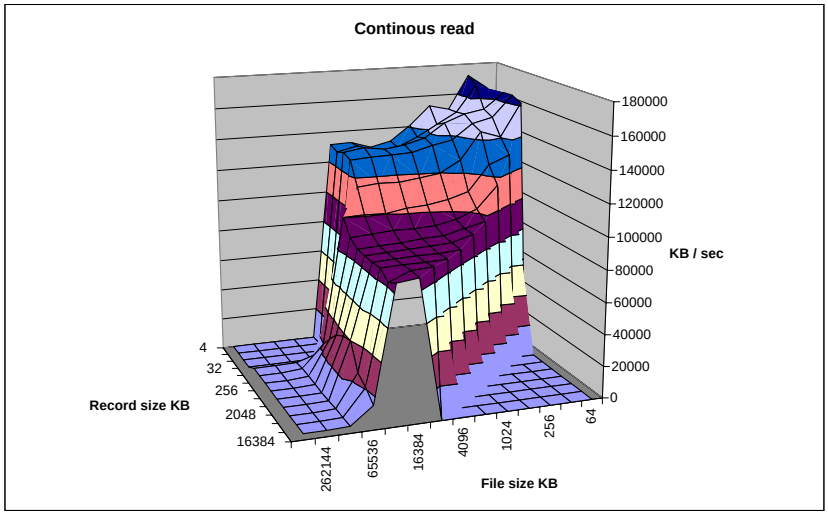

Figure 2.8: Linux continuous read

improves the performance as data is read from the buffer and not the disk. Please note that the usefulness of the cache is strongly dependent on the scenario of usage.

Write cache When the destination for data is slower than the sender there is a performance advantage of storing the data on a faster buffer for retrieval by the slower destination. The sender can thereby continue operations. This is also called delayed write and might also increase throughput on the disk by writing continuous blocks and letting newer writes negate earlier change in data.

disk Cache may be used to overcome some performance problems from disk.

2.4.2.1 On-disk cache

Because it takes time for the internal mechanics of a disk drive to move, the access time and throughput suffer. The throughput is limited by the density and the rotation speed of the platters. Latency is limited by the speed of the heads.

On-disk cache is usually 64KB-8MB. Smaller disk caches <256KB is mostly a buffer to compensate for the difference in speed in the disk interface and the physical disk. Only larger disk caches 256KB might give cache hits. The interface for the disks, ATA or SCSI limits the throughput from the on-disk cache to 100 or 320MB/s respectively.

2.4.2.2 Cache size

If An is the memory area used by client n and m is the number of clients then the area for the buffer cache S to cover has the size of:

$$S = \bigcup_{n=1}^{m} An \quad \leq \quad \sum_{n=1}^{m} An$$

Buffers may be used on both clients and servers. Buffer cache on the server computer(s) have the highest utilization as more requests go through it. The need for a bigger cache as the data size increase creates a need for more than one server. Cache on the clients scale better in size but does not scale past a certain number of clients if strict consistency is to be kept. To maintain strict cache consistence between the clients the communication have to be frequent and coordinated between *all* the clients.

Cache hits give performance boosts so bigger cache and more intelligent cache handling give better performance. The size of the needed cache increases with the size of the data being accessed.

Figure 2.9 show the impact of cache size. The raw disk throughput through the file system is here about 4-5MB/s which is a cache size. The raw disk throughput is here about 4-5MB/s which is a fraction of the maximum throughput of the disk. The reason for this is possible unoptimized disk settings and fragmentation. The cache is visible as higher throughput for files smaller than 32MB. The performance with cache hit is unaffected by the disk throughput. The Ultra 1 shows here in figure 2.10 better throughput than the Linux box for file sizes larger than 32MB.

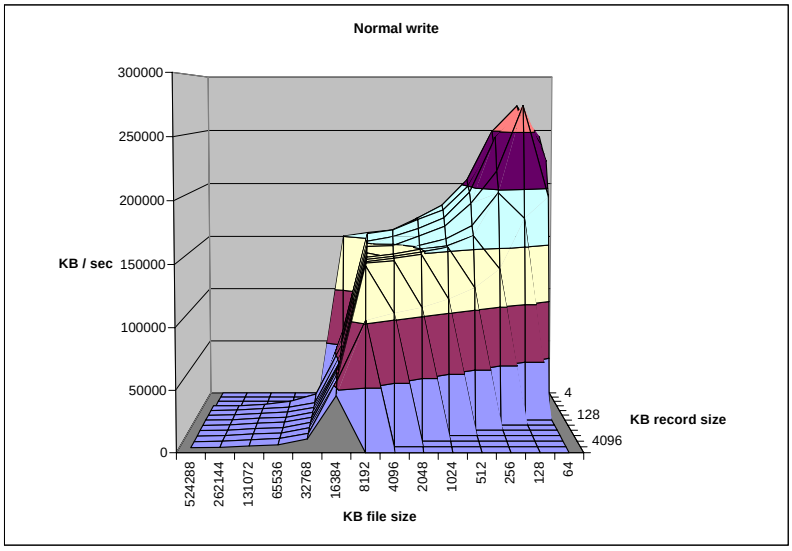

Figure 2.9: Linux continuous write by file size

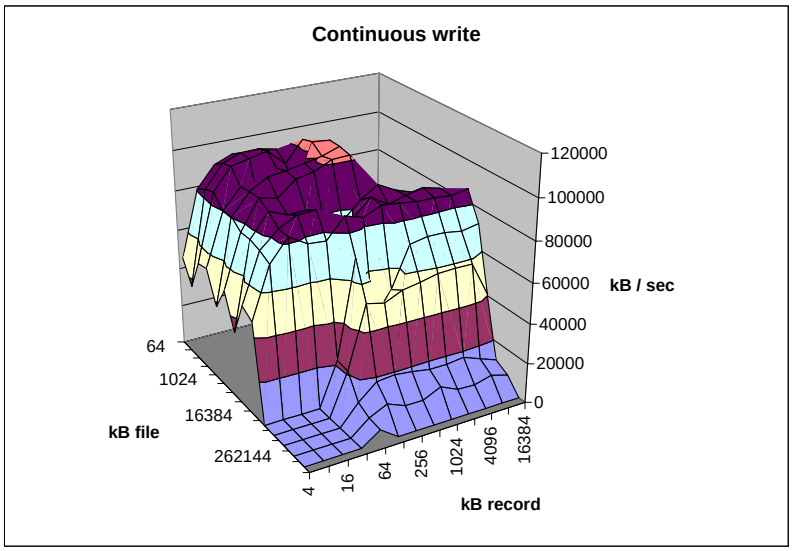

Figure 2.10: SunOS continuous write

2.4.2.3 Cache prediction / prefetching

A database has a mainly random access pattern and thereby removes much of the performance gain from cache prediction and prefetch as the record size is often quite small. Oracle uses a record size of 8KB.

Handling of ordinary files is often done in a continuous access pattern. Here prefetching and cache prediction have considerable significance.

In an environment of only read, buffering both in server and client memory is possible and increase the performance a lot.

2.4.2.4 Cache influence

The influence of the different caches in the system is shown in figure 2.8. The peak performance is at 170MB per second. The physical throughput is here about 4MB per second. This means that for small files that are cached completely, performance is 42 times better because of the cache. Figure 2.11 shows the impact of a reduced memory cache.

Canceling writes It is interesting to notice that performance is better for continuous writes, see figure 2.9 and 2.11, than for continuous read, see figure 2.8. This is mostly for file sizes less than 512KB but is also visible for larger file sizes. This is because a cached write that is changed again in the cache before synchronized with the disk means that the first change does not have to be pushed through the rest of the I/O pipeline to the disk which have the least bandwidth. This also have the side effect of making the performance less dependent on the record size since the later parts of the pipeline that handles 4KB buffers is not involved. See section 2.3 for the explanation of the record size impact in the pipeline.

2.4.2.5 Different cache makes plateaus

Figure 2.8 shows the different plateaus quite well. First level cache contributes to the higher performance for file sizes 256KB or less. Buffer memory gives the higher performance for file sizes less than 32MB. The actual physical throughput is the limit for files larger than 16MB.

There are different levels for reads and writes. This is because the I/O pipeline is different. Below I have shown the different plateaus from the test results from the Linux workstation as this had the most evident plateaus.

2.4.2.6 Plateaus for reads

The following is seen in figure 2.8 and

2.6. The first plateau is for file size of 64KB to 128KB size. This is the L1 and L2 on chip caches. The next plateau visible is at 256KB file size and record size of 64KB or less. This is L2 on chip cache. The third plateau is file sizes less than 64MB. This is conventional memory used as buffer. The fourth plateau is for file sizes larger than 64MB. It shows the physical throughput of the disk and to a smaller degree the bus.

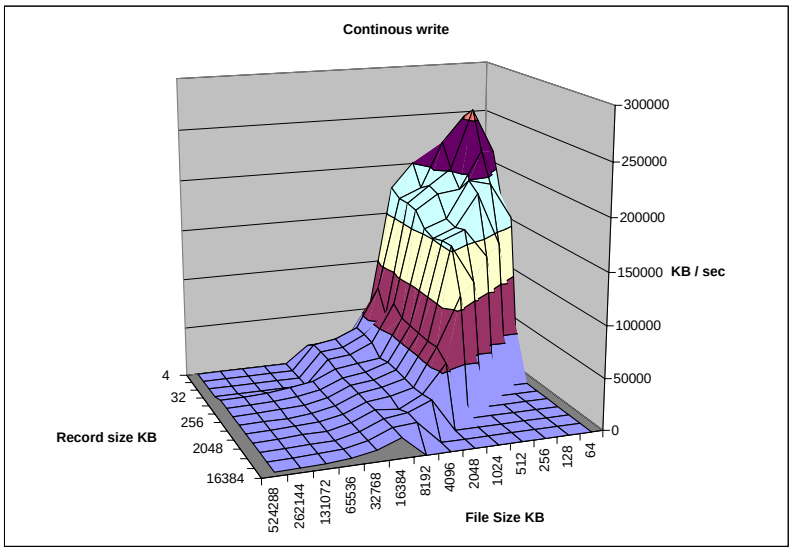

Figure 2.11: Linux continuous write with low memory

2.4.2.7 Plateaus for write operations

The following is seen in figure 2.11 and 2.9. The first plateau is file sizes less than 128KB. This is the L1 on chip cache. The second plateau is less visible but is file sizes less than 256KB. This is the L2 cache. The third plateau is for file sizes less than 8MB and is the use of conventional memory as buffer. This is quite different from the read tests and shows that the write pipeline has less buffering than the read pipeline. One reason for this is that buffered write operations might give inconsistent data. The amount of memory available is also very significant. A rerun of the tests on the same computer showed the memory buffering plateau to be for file sizes less than 32MB. When the second test was run the amount of free memory on the box was larger than when the first test was run. Since the Linux kernel version that was in use during the tests uses most of free memory as disk buffers this is not unexpected results. It also shows the impact of the size of the buffer.

2.4.3 Log-structuring

In log-structured file systems such as described in [RO91] which utilize cache, higher performance is achieved by serving most reads from the cache while writes are done in an effective manner by writing chunks of continues data. Test results in [RO91] showed that the log-structuring enabled 70% of disk bandwidth for writing, whereas Unix file systems typically can use only 5-10%.

Thus this approach is slower on sequentially reads after random writes when the data is not cached.

The underlying structure of a Log-structured File System (LFS) is that

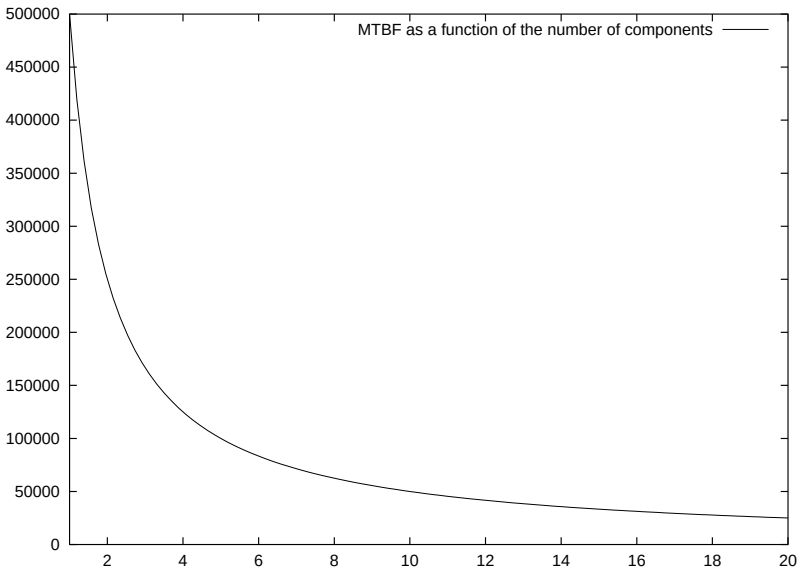

Figure 2.12: MTBF as a function of number of components

of a sequential, append-only log. In ideal operation, all log-structured file systems accumulate dirty blocks in memory. When enough blocks have been accumulated to fill a disk track, they are written to the disk in a single, contiguous I/O operation. All writes to the disk are appended to the logical end of the log. This is a delayed write.

Although the log logically grows forever, portions of the log that have already been written must be made available periodically for reuse since the disk is not infinite in size. The process is called cleaning.

This technique is similar to the later Write Anywhere File Layout (WAFL) technique [HLM94] that is to be used in Reiser4 [MRZ02].

2.4.4 Disk failure

Disks are prone to errors. Failure is inevitable. There may be a long time between errors, but errors will appear. Errors destroy data. Edward A. Murphy is often cited with the words "If anything can go wrong, it will". Disks are the only moving parts in a computer today except for ventilation fans and CD/DVD drives. They are therefore a bottleneck in hardware stability.

The Mean Time Between Failure (MTBF) for a component might be 500 thousand hours which is about the best for desktop computers today, but it will eventually fail. With a system with two system critical components with MTBF of 500 thousand, the MTBF of the systems become 250 thousand. Thus the MTBF decrease as more components is added. Equation 2.4.4 show how MTBF is calculated and figure 2.12 show the total MTBF for a system of 1 to 20 system critical components with MTBF of 500 thousand each.

$$Total\ MTBF = \frac{1}{\sum_{i=1}^{m} \frac{1}{Component_i\ MTBF}} \tag{2.4}$$

	ReiserFS	Ext2
I/O		
Cont. read 10MB file	185.0MB/s	183.3MB/s
Cont. write 10MB file	187.9MB/s	178.7MB/s
Rand. read 10MB file	171.0MB/s	180.1MB/s
Rand. write 10MB file	171.5MB/s	169.7MB/s
Cont. read 1GB file	24.7MB/s	23.7MB/s
Cont. write 1GB file	24.6MB/s	25.0MB/s
Rand. read 1GB file	9.8MB/s	9.5MB/s
Rand. write 1GB file	6.7MB/s	6.6MB/s
Meta data oper.		
Rand. fseek 10MB	12.83us	12.86us
Rand. lseek 10MB	0.80us	0.82us
Open/close empty file	2.64us	2.62us
Create/delete empty file	86.29us	8.17us

Table 2.3: Impact of Journaling

Where: *Total MTBF* is the resulting MTBF
m is the number of components
Component MTBF is the MTBF for each component

2.4.4.1 Handling of disk errors

Traditionally, such recovery is handled by tape backups only. More and more common today is the use of simple mirroring or RAID to deal with disk errors, often in addition to tape. Implementations of RAID 5 may recover so gracefully from a disk crash that no downtime is registered at all. The latest price development for tape cartridges and hard drives show that hard drives are actually cheaper than tape cartridges. This means that backup on disk might be preferred. But the backup disks should not be located in the same place or building as the one with the production data. There is little point in a backup if it is destroyed in a fire together with the disk.

2.4.5 Journaling file systems

Journaling file systems use an idea taken from the world of databases. At checkpoints the file system writes its entire state to disk. It then writes all changes after that point both to its log and to the file system itself. The log is rolled forward after a system failure to ensure that all the changes requested after the checkpoint are reflected in the file system.

Thus it is possible to roll the file system forward to a consistent state without rechecking the hole device.

There are two types of journaling file systems. The first type uses a journal as an adjunct to an existing file system. The second type implements the entire file system as a journal or log, a so called log-structured[RO91]. There is more about log-structured file systems in section 2.4.3.

Table 2.3 show the overhead of journaling by comparing ReiserFS and Ext2. This table also show the importance of prefetching and access time to disk. Continuous operations on a 1GB file show about 3× the performance of that

of random operations. An other interesting result is that the stream interface for files have much slower seek than with file descriptors. Meta data operations tested are open/close, create/delete and fseek, lseek. It might be argued that more separate testing by for instance dividing testing of open and close would give more interesting results but to achieve somewhat average results the number of files needed in separate open and close testing would lay extra strains on the file system and therefore influence the test results.

The computer used for this tests had 256MB of RAM. ReiserFS is considered a fast journaling file system while Ext2 is so simple in its construction that further enhancements are difficult. A comparison of these two might therefore give an indication of how well a journaling file system can perform compared to a fast non-journaling file system.

All I/O operations were done with 256KB record size. The actual numbers should not be taken as a performance test for the respective file system but more as a comparison between them. Both file systems were installed on the same disk in the same computer. They were also on the same partition since hard drives is slower for the highest sector numbers than the lowest. This is because the lowest sector numbers is placed at the rim of the plattersåwhere data pass the heads much faster. For the 10MB file sizes this probably have no importance. The computer used for this test had 256MB of RAM.

The differences in I/O performance is about 7% faster for Ext2. fseek and lseek is 2.5% and 0.2% faster with ReiserFS respectively and open/close is about 0.8% faster for Ext2. For create/delete Ext2 is 90.5% faster than ReiserFS. Since create/delete is much less used than the other operations the test results show that a good implemented journaling file system might perform equally good as one without a journal. The reason for ReiserFS to actually have better performance in seek operation is due to the use of fast balanced trees.

2.5 Memory to CPU

Memory is much slower than the CPU internal bandwidth. Again cache in a faster medium than the slowest is the major improvement. Note that it is only sensible to increase performance of memory if it is a bottleneck. In figure 2.11 disk is not a bottleneck for data sets under 1KB even if the lack of memory cache decrease performance for larger data sets.

2.5.1 CPU cache

Today there are 2 or 3 levels of extra cache consisting of faster memory between the CPU and the main memory. Level 1 is on-chip while level 2 may be

on-chip or on-board. Level 3 is almost always on-board. As each level is closer to the CPU is faster but more expensive to manufacture and therefore smaller. A typically level 2 cache is today 256-512KB. The computer in figure 2.9 shows about twice the throughput of main memory throughput during peak performance because of this extra CPU cache.

CPU cache is faster than the rest of the data pipeline by $\times 10$ and this gap is increasing with technology improvements. It is therefore not much that can be done to increase the overall I/O performance through improvements of this cache. Instead another approach is used. Using the CPU for transferring data to

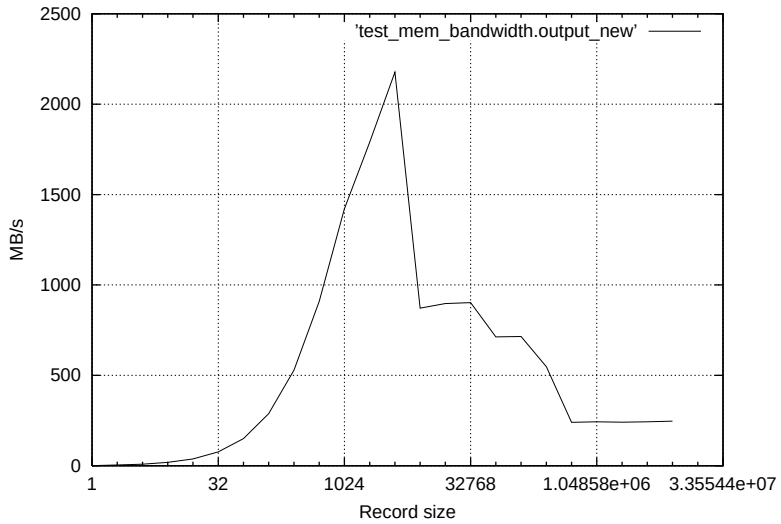

Figure 2.13: Memory and cache bandwidth by record size

and from peripherals is very inefficient as the CPU is then unable to do anything else during the transfer. The invention of Direct Memory Access (DMA) enabled the devices to bypass the CPU, allowing the CPU to do other work and let the peripherals transfer data between themselves, leading to increased performance. Today the throughput for DMA is limited by the PCI bus that transfer the data. Table 2.1 show this bus can be as fast as 486MB/s.

2.5.1.1 Record size

Record size is important to the performance of continuous read operations. This is clearly shown in figure 2.6 and 2.8. Record size does not affect the cache hit rate but it strongly affects the efficiency of the pipeline as described in section 2.3. This is even more visible in figure 2.13 where 4KB record size gives the peak performance but 8KB gives under half the performance at 4K. The test were conducted by repeatedly copying from the same address and thus maximizing CPU cache hit rate.

2.5.2 CPU

The CPU is faster than the rest of the data pipeline by an order of magnitude and this gap is increasing with technology developments. Because of this the choice of CPU give little if any impact on I/O performance.

2.6 File systems and ACID

Most of all the ACID attributes in a storage system, be it a file system or a database, improve the handling of these issues (ACID) in the applications that use the storage system. Handling these issues in the storage system lower complexity and often increase performance. But it is important to remember that they take time and if they are not needed they should be avoided.

Streaming a video stream directly to a raw disk device onto continuous blocks is not atomic, concurrent or isolated, but it is by far the fastest approach to write large amounts of data to a disk.

Atomicity ensures that an operation is either completed or undone to avoid inconsistency. In a file system journaling have been used to ensure atomic changes of file system metadata to avoid inconsistency. For changes of data in different files to become atomic as a whole, a more extended approach is needed as in Cedar File System (CFS) or Reiser4.

Concurrency is the ability to run simultaneous operations to increase performance. Multiple threads or job queues handle this efficiently.

Isolation allows different threads and processes to operate simultaneous on the same data with the results of the operations being similar to the operations being performed separately in time. For this locks are the most used in file systems while databases with rollback also use timestamps. There as far as I know of no examples of handling of dead-locks and live-locks in file systems as this is left to the applications.

Durability is persistent storage and ability to retrieve the data later. Journaling and misc. file system repair programs ensure data correctness. Disks, tapes, CD-ROMS, and DVD are the most used media.

2.6.1 Atomicity in file systems

Journaling file systems of today like Ext3, JFS, ReiserFS and XFS all have the ACID attributes except atomicity.

Atomicity have long been used in databases and have improved both ease of programming and performance. Databases have been highly optimized to handle parallel access. It is difficult to beat them in this game. Introducing atomicity to file systems might transfer this as well.

In traditional Unix semantics a sequence of write() system calls are not expected to be atomic, meaning that an in-progress write could be interrupted by a crash and leave part new and part old data behind. Writes are not even guaranteed to be ordered in the traditional semantics, meaning that recently-written data could survive a crash even though less-recently-written data does not survive. Some file systems offer a kind of write-atomicity, known as data-journaling as described in section 2.4.5. But this only ensures that individual blocks or the buffer of a write() system call are written atomically. For entire files or file systems writes are still not atomic.

The CFS presented in [GNS88] is a file-level-access system. Its main characteristic is that remote files are immutable similar to the earlier swallow [RS81]. This is used to make updates atomic in the Distributed File System (DFS) system that uses the CFS to obtain consistency in the subsystems of several files. This system is *not* UNIX POSIX compliant but introduces its own file semantics. Performance in such a system should be able to compete with an ordinary file system with the use of clever (i-node) handling where

unchanged blocks points to the original copy with counters in the same way as hard-links is implemented in a standard *NIX system.

Journaling is implemented in two ways, either the data is written first in the journal and then in the file system or the block is written only once to a new location. Then the block's address in its parent node in the file system is updated. This increase speed but the use of pointers also allow atomicity of a complete file system. A method for letting the parent modification be included in the transaction is the WAFL technique [HLM94] which handles this by propagating file modifications all the way to the root node of the file system, which is then updated atomically. This is implemented in Reiser4 [MRZ02] which is expected late 2002.

2.7 Concluding remarks

Prefetching and caching data is essential for performance. Cache hits can improves performance by a factor of $> 50\times$ for that of a single disk. Prefetching con improve performance by $20\times$. More memory for buffering means better parallel performance. Canceling older writes in the cache improves performance when data is changed rapidly.

More I/O-threads in a systems adds to the randomization of the operations. This leads to more difficult prefetching and the need for larger cache.

Cache is crucial for performance. Server cache with server striping scales better than client cache in a read-write environment. In a read-only environment client cache scales better than server cache.

Chapter 3

File systems for clusters

The environment in focus for this thesis uses a cluster topology, MPI and SCI for high performance computing and thus need a high performance distributed file system. Chapter 2 explained the local part of the I/O path. This chapter uses this knowledge and evaluates different distributed file systems for suitability in the environment described.

Large tasks are divided into smaller tasks and distributed to individual computers for faster processing. The kind of task at hand decide how tightly integrated the system of individual computers should be. Different systems have altogether different needs. This is mainly because the loosely coupled ones share data in a less degree than the tightly integrated ones. In this chapter the acronyms Highly Integrated Parallel System (HIPS) and Loosely Coupled Parallel System (LCPS) will be used to name such systems in a convenient way.

3.1 Distributed file systems

The following paragraphs describe file systems best suited for HIPS and LCPS and compare them. No system excel in both HIPS and LCPS.

3.1.1 NFS

Sun NFS was designed by Sun Microsystems in 1985 [SGK+85] and is one of the most known distributed file systems in use today. Especially on Unix compliant platform (*nix). Its widespread use and simple client server architecture have made it a reference for many tests done to distributed file systems.

The simple implementation also limits performance since all data have to be fetched from disk on a central server every time. Later versions use cache on the server and clients to increase performance. It is also important to note that in a multi-user environment accesses from many simultaneous users increase total randomness which is bad for disk performance even if each user generate sequential operations. Most applications access data in a sequential manner and this means that cache is more important than ever because of the long disk access times.

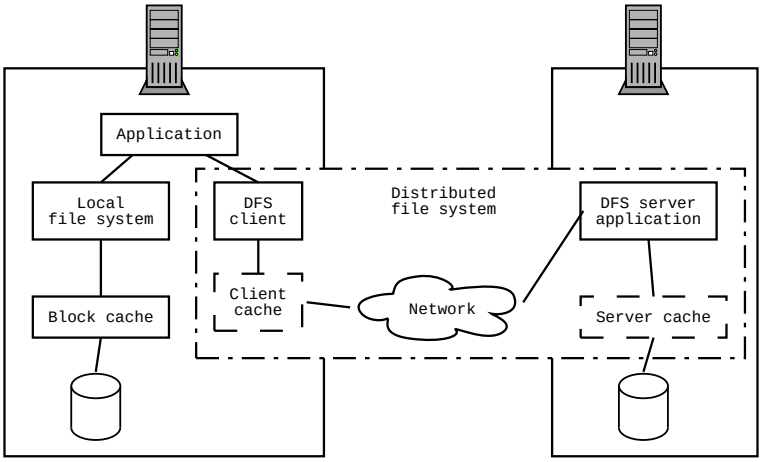

Figure 3.1: Local and distributed file system

3.1.1.1 Server cache

The server cache works in the same way as the local cache as explained in section 2.4.2 to increase the throughput and access time from disk. Figure 3.1 show the schematic placement of the cache. Note that the server cache might be part of the DFS server application. Since all the queries from all the clients goes through the server, this gives better hit-ratio than client-only caching.

3.1.1.2 Client cache

In the beginning of chapter 2 it was stated that a distributed file system can never be faster than a local file system because of the longer and narrower I/O pipeline. Caching on the client creates a chance to avoid accessing the server altogether. According to table 2.1 local cache makes the I/O throughput 10× faster than server cache when there are cache hits because the difference in speed between network and local memory. A client cache does not exclude the use of a server cache.

Figure 3.1 shows that the I/O pipeline with hits in the client cache is equally short as the local one when the data direction is from server to client. In the other direction, delayed write is used. This enables grouping of writes and one write to nullify an earlier write to the same data and thereby reducing the needed bandwidth.

The problem with this is consistency between clients when there are multiple copies of the same data in the client caches. The NFS solution to this is to not guarantee propagation of changes before 6 seconds have elapsed. This avoids purging the caches too often and in most cases provides reasonable consistency.

3.1.1.3 Architecture

NFS uses stateless client server architecture. If a server is rebooted the clients will experience this as a slow response. In later versions this can be changed

dependent of mount options. Scalability is achieved by the system administrator by splitting data directory-vise on different servers.

3.1.1.4 Drawbacks with NFS

NFS does not scale well for many clients performing simultaneous read and write access to the same data (hot spots). With HIPS there are often many such hot spots and NFS will therefore not scale well. Consistency in NFS is only acceptable for HIPS when the client cache have been switched off either totally or when using locking like flock() and fcntl(). Such locking was not part of the originally NFS design. NFS uses a block addressing scheme relative to the start of the file. This is different from most OS's where the addressing is done relative to the start of the volume / partition, and there is some performance loss. See section 2.3 for further explanation of this. NFS also strain write performance since servers must store data safely before replying to NFS write requests. There have been NFS implementations with non-volatile RAM to reduce NFS response time.

NFS have an addressing scheme that is file oriented. This means that addresses is part filename and part address within the file. This has implications on caching. Olaf Kirch, the author of much of the NFS code on Linux, have stated that it would be very difficult to use the Linux block devices cache with NFS. Therefore the client cache in NFS have been implemented separately.

3.1.2 Zebra

In the previous section the problems with hot spots and scalability were described. Zebra [HK93] tries to solve this by a log-structured file system striped across servers like RAID [DAP88]

This file system is important because it was the first (at least that I know of) file system to act as a conventional file system while striping data between servers. The earlier Swift [CD91] also used distributed disk striping in a similar way but the file system layer was not present. Since this was an early approach, important aspects was discussed while developing this system that was not repeated in papers of later systems that use the same technique.

The general idea is to do striping across multiple storage servers, with parity, similar to RAID levels 2 and higher. This helps with:

- load balancing - load is distributed among the storage servers, which avoids the problem with hotspots described for NFS.
- parity for availability - if a server goes down, the remaining ones can be used to recover the missing contents.
- higher throughput - max aggregate bandwidth is the sum of the individual throughputs of each server. This allows for more and more demanding users.

Striping is not done per file, which would limit the benefit (since the file would need to be chopped to tiny pieces), but per log segment. This globbes writes together, like the earlier LFS [RO91] described in section 2.4.3. Per-file striping causes a discontinuity in the way small and large files are handled. If, on the other hand, small files are stored in fewer servers than the full array,

then the space overhead of parity becomes huge. Here, logging is used between a client and its file servers. This is opposed to the original LFS case, where logging was used between a computer and its disk. Each file system client uses its own log and stripes its log segments, instead of the individual files.

File meta-data are managed centrally by a file manager. The logs contain only data blocks. The file manager also deals with cache coherence, and name space management. It has to be contacted for all open and close operations. It has a local "shadow" file for each file accessible through the Zebra file system. The local shadow file contains block pointers and similar. Deltas contain meta-information that clients write into their logs; they don't contain data. However, deltas are also read by the file manager, so it can manage meta-data accordingly.

Segments are reclaimed by a stripe cleaner, similar to a segment cleaner in LFS. Instead of using segment summaries, as in LFS, the cleaner keeps track of usage information in stripe status files, which it keeps as normal Zebra file system files.

Recovery in such a system is more complex due to the distribution of tasks. In addition to ordinary failures it have to handle these extra failure cases:

- Client crashes - in which case, a partially stored stripe might be regenerated from a parity fragment, or discarded
- Storage server crashes - in which case, either a fragment has been partially stored or some fragments should be stored but haven't. Here checksums help find incomplete fragments and parity helps reconstruct missed fragments
- File manager crashes - meta data might not be up to speed given the current deltas; delta replay is used to bring the meta data up to speed, assisted by checkpoints of covered deltas per client
- Cleaner crashes - in which case, the cleaner status for each client's stripes might be out of date; log replays are used here as well.

Meta data in a file system is file names, create date, change date, permissions and locks among others. Remember also that the MTBF decreases with increased number of components.

The only critical resource is the file manager. Any other single crash by another component can be dealt with on-line, without reducing availability. Recovery of any single failure can occur while the system is running.

3.1.2.1 Server cache

With RAID 0 server cache is equally easy as without striping since there is no duplicate data. With striping types which include redundancy coordination to maintain cache coherence is needed between the servers.

3.1.2.2 Potential disadvantages:

- Small files are only collectively benefited. That is, if you're only dealing with few small files, you still have the problem with LFS not being the best paradigm to use.

- Striping introduces a higher error rate, more disks and servers that can fail, see section 2.4.4. The error rate is then reduced back by parity striping units. It's unclear whether there's a net decrease in error rate.
- The larger the number of servers, the larger the stripe size has to be to make the fragment size efficient, i.e. more buffering is needed than in the single-host case. That's why stripe groups where introduced in the evolution of the Zebra file system [ADN+95].

Streaming data across several servers like it is incorporated in Zebra have later been incorporated into xFS [ADN+95], Tiger Shark [HS96], on the Intel Paragon [Aru96] and PVFS [CLRT00].

3.1.3 xFS

It is important to note that xFS [ADN+95] is not the same as XFS [SDWH+96] which was developed by SGI and ported to Linux is a local file system. xFS was made at Berkeley and is a distributed file system.

Zebra [HK93] is not completely decentralized. The general idea of xFS is: ZebraFS completely decentralized. Any participant can take over the role of any failed component. Also, xFS employs cooperative caching, using the participant's memory as a global file cache. It is based on a Network Of cooperating Workstations (NOW), whose kernels trust each other completely. This have with success been used as a fast NFS server.

Improvements from Zebra:

- Scalable and distributed meta data. Scalable and distributed cache consistency management with reconfigurability after failure.
- Scalable sub grouping to avoid excessive distribution.
- Scalable log cleaning.

Meta data are handled by managers. A shared, globally replicated manager (map) assigns a meta data manager to a file. The assignment of files to managers is done through assigning different bit areas of the inumber to different managers. New files are given appropriate index numbers so they are managed by a manager living on the same node that created them.

Each meta data manager has an IMAP to locate the blocks to its files. In fact, an IMAP entry contains the disk location of the file's i-node, which contains the block addresses or the addresses of indirect blocks.

Storage servers are split into stripe groups. A stripe is split among fewer storage servers than all those available, to make sure that stripe fragments don't turn too small or, conversely, that clients don't have to buffer enormous amounts of data. Each stripe group has its own parity. Therefore, simultaneous storage server failures in different groups can be recovered from. Also, clients can write to different stripe groups at full speed at the same time.

i-nodes are only cached at the managers, so that the manager doesn't get bypassed when a local cache can't satisfy a request. This was decided so that cooperative caching could always be preferred to disk access. Caching is done per file block, not per file and follows the write-invalidate protocol.

Recovery is accomplished through check pointing, roll-forward and distributed consensus for membership list recovery.

3.1.3.1 Potential disadvantages

As with Zebra, small files are only collectively benefited. xFS was a research project and as the source code have not been made totally available due to the license restrictions on parts the system were integrated with, there have been no further development of xFS.

3.1.4 GFS

Computer failure makes the local disks unavailable even if the data is consistent and the disks themself work fine. The aggregated throughput for disks that work together in a RAID may be more than one computer can handle. Thus the server is a bottleneck and a single point of failure. The Global File System (GFS) [SEP+97] solves this by connecting the disks and the clients to a shared interconnect and thereby eliminating the server. The disks have very little cache. This is compensated by cache on the clients.

GFS distributes the file system responsibilities across the clients, storage across the devices, and file system resources across the entire storage pool. Data is cached on the storage devices and clients. Consistency is established by using a locking mechanism maintained by the storage device controllers to facilitate atomic read¡modify¡write operations. GFS is accessed using standard Unix commands and utilities. The GFS clients have with success been used to export the GFS volumes as NFS volumes.

3.1.4.1 Shared disk or shared file system

Shared disk or shared file system as a *cluster file system* is evaluated as two different approaches in “Evaluation of Design Alternatives for a Cluster file System” [DMST95]. Performance measurements of caching efficiency, scalability, throughput, and write-sharing showed that the shared file system two advantages: about 2 to 4 times better performance in write-sharing and about 30% better read throughput. The shared disk approach, however had a small advantage when almost all accesses can be serviced by the local file cache. Measurements also indicated that, in the shared-disk approach, multi-ported disks did not improve write sharing performance unless the disks used `Non Volatile RAM (NV RAM)` caching. The overall conclusion was that the shared file system approach is the right direction for a cluster file system.

In spite of the implications of this statement GFS are using the shared disk approach. In the first GFS paper [SEP+97] there were also stated that the portability of shared file systems, or “shared nothing” as the paper called it, is better than that of shared disk. Arguments were given in [SEP+97] that networks and network attached storage devices had advanced to a level of performance and extensibility that the once believed disadvantages of "shared disk" architectures were no longer valid. This shared storage architecture attempts to exploit the sophistication of device technologies where as the client¡server architecture diminishes a device's role to a simple components.

Disk cache improves both read and write performance with read-ahead and write behind.

3.1.4.2 Coordination in client cache

As described in secion 3.1.1.2 client cache increase I/O throughput more than $10\times$ than server cache when there are hit. To avoid different versions of the same data in multiple client caches a disk based lock scheme is used. Locks resident on the drive are used by multiple clients to safely manipulate file system metadata. Atomic operations on shared data are performed by acquiring exclusive access to the data via a lock, writing the data and releasing the exclusive access by giving up the lock.

To do this most efficiently a new SCSI command called DLOCK [Sol97] was developed. This device locks (DLOCKs) are implemented as an array of state bytes in volatile storage on each device. Each lock is referenced by number in the SCSI command: the state of each lock is described by one bit. If the bit is set to 1, the lock has been acquired and is owned by an initiator (client). If the bit is 0, the lock is available to be acquired by any initiator. The DLOCK command action test and set first determines if the lock value is one. If the value is 1, then the command returns with status indicating the lock has already been acquired. If the value is 0, DLOCK sets the lock to 1 and returns GOOD status to the initiator. The DLOCK command clear simply sets the lock bit to 0. A test operation is provided to read (but not set) the state of the lock. The locks is set to cover a 4KB or 8KB block. on the disk.

Newer versions allow the use of an external distributed lock manager but the handling is essential the same.

This is similar to having the server in a client-server approach storing the locks. It is different from ordinary cooperative cache as described in [Cor97] since the clients only communicate with the device and lock manager and not between themselves. Standard cooperative cache have in worst case communication in n^n directions at once to maintain cache coherence. This congests for n larger than $8-16$, even if the network can handle it the computers themselves will be too occupied.

3.1.4.3 Potential disadvantages

The DLOCK approach in GFS have two drawbacks versus locking on server:

1. Acquire of lock and I/O operation demands a lot of network accesses. One for each lock and one for I/O.

2. When a lock is acquired other clients drop their cache of the same data. Then when they get the lock there is almost no buffer cache since on board disk cache is at most 8MB. Throughput therefore decreases.

3. GFS was open-source but became closed. This might be an obstacle for further research.

GFS is further discussed in section 3.2.1 where it is compared to PVFS as a HIPS cluster file system.

3.1.5 PVFS

Most DFSs are not designed for high-bandwidth concurrent writes that HIPS typically require. PVFS [CLRT00] focus is high aggregated bandwidth for concurrent read and write operations to the same data. There have been measured aggregated 1.5GB/s and 1.3GB/s for aggregated read and write throughput respectively [Bae02].

PVFS is intended both as a high-performance parallel file system that anyone can download and use and as a tool for pursuing further research in parallel I/O and parallel file systems for Linux clusters. The source code of PVFS is licensed under the General Public Licence (GPL) and it is therefore very easy to do changes to the system.

3.1.5.1 Striping across servers

Data is striped across multiple servers and thereby balancing load on servers and network connections. Performance scale well by adding more servers [Bae02]. The difference from Zebra is how data is striped. Striping is done per file in 64KB chunks to ensure that files are not cut to too small pieces. There are no redundancy in the stripe data and therefore there is just write and not read-modify-write as in the parity schemes. This increase performance for small writes.

3.1.5.2 Multiple APIs

PVFS support multiple API's. A native PVFS API, the UNIX/POSIX I/O API [pos] as well as other APIs such as The I/O chapter of MPI (MPI-IO) [mpi97, GLT99]. Thus applications developed with UNIX I/O API can access PVFS files without recompiling with the use of preloading. The native PVFS API is faster than the UNIX/POSIX I/O API. Performance when calling through the kernel is about $\frac{1}{60}$ of the performance when calling directly in user space with the native functions. The UNIX/POSIX API through the kernel provide the ability to use common UNIX shell commands such as ls, cp and rm.

The most used programming environments for developing on clusters are MPI and Parallel Virtual Machine (PVM). They cover the communication between the different parts of the application but have until recent years not had any integrated way of handling persistent data. The introduction of MPI-2 with an I/O chapter MPI-IO changed this.

ROMIO is a portable freely available almost complete implementation of MPI-IO and it supports PVFS. It is optimized for noncontiguous access patterns, which are common in parallel applications. It has an optimized implementation of collective I/O, an important optimization in parallel I/O. ROMIO 1.0.1 includes everything defined in the MPI-2 I/O chapter except shared file pointer functions, split collective data access functions, support for file interoperability, I/O error handling, and I/O error classes.

3.1.5.3 Drawbacks with PVFS

Meta data operations is slow under PVFS. It is therefore not suitable for systems with many small files. More on the implications of this in section 3.2.1.

3.1.6 (AFS) / OpenAFS

As described in section 3.1.1.2 client cache increase I/O throughput more than 10× than server cache when there are hit. Caching locally also removes load from the server and thereby increasing scalability. In most environments, simultaneous operations on the same date is very rare. The cache coherence overhead is therefore small. These three points makes local disk and memory cache very preferable in most environments. AFS and OpenAFS, hereby called *AFS, utilize this by using extensive local cache. *AFS is designed to scale to more than 5000 machines.

AFS is a distributed file system product, pioneered at Carnegie Mellon University and supported and developed as a product by Transarc Corporation (now IBM Pittsburgh Labs). It offers a client-server architecture for file sharing, providing location independence, scalability and transparent migration capabilities for data. This was a part of the Andrew project and this was the Andrew File System (AFS), but it have since then evolved beyond the Andrew project and the acronym no longer yields.

3.1.6.1 OpenAFS

IBM branched the source of the AFS product, and made a copy of the source available for community development and maintenance. They called the release OpenAFS. It's available from the IBM Open Source site. The openafs.org site is for coordination and distribution of ongoing OpenAFS development.

Read and write have a granularity of whole files or partial files if they are very large. *AFS therefore needs disk on local machine to handle reasonable size files. This is also implemented in Cedar [GNS88, RS81].

The synchronizing schemes used for the client cache have also been implemented between servers and thereby providing multiple servers for the same data. This increase both scalability and performance.

3.1.6.2 Drawbacks

The efficiency is degraded with small updates in many files. Coherency is also worse than with *block-level-access*. Simultaneous access to the same data turn the client cache into a performance slow-down instead of a booster. In general write operations increase the replication work and therefore decrease performance and scalability.

3.1.7 Coda

Mobile computing demand synchronizing of data whenever connected. Carnegie Mellon University's Coda [Sat90, SKK+90] makes this easier by having a more persistent cache on the client, it is possible to access the files when not connected to the network and automatically synchronizing whenever connected. This also provides continued operation during partial network failures.

3.1.7.1 Constant availability

Coda (Constant data availability) was designed to improve the availability of the Andrew file system. As *AFS it has replication between servers. A server

failure may then have little impact on availability. This approach also allows clients to run in disconnected operation using only the files it has cached locally. The client can reconnect to the network and synchronize its cache with the rest of the system. Like the Sprite file system [NWO88], Coda servers maintain state concerning file accesses. The servers are responsible for performing call backs when a clients cached data has been modified by another client. File sharing on a client is guaranteed to have consistency described by Unix file sharing semantics. Files shared across different systems see consistency as the granularity of the entire file.

3.1.7.2 Drawbacks

Mostly the same as with *AFS and disconnected operation increase possibility of consistency mismatch.

3.2 Discussion

This section is dedicated to comparisons and more in-depth evaluation of technology in the setting of the existing implementations and HIPS.

3.2.1 GFS versus PVFS

Both are newer file systems. There was first published something on GFS in 1996 and for PVFS in 2000. They can use striping across multiple storage mediums and their target areas does include scientific computing.

The main difference between PVFS and GFS is how the storage disks are connected to the clients. In GFS the disks and clients are directly connected to a shared interconnect. In PVFS the clients are connect to servers that have the disks connected locally. GFS uses fiber channel or SCSI as the shared interconnect. PVFS uses Ethernet, Myrinet or SCI as supplied by this thesis. GFS have means of fail tolerance and PVFS have none. GFS is considered a conventional file system and PVFS is optimized for tightly integrated parallel programs.

3.2.1.1 I/O path

The direct connection of the disks in GFS makes the I/O-path shorter and therefore the latency shorter compared to PVFS. Throughput on the other hand is not affected as shown in section 4.5 and explained in section 2.3. The use of cache to shorten the I/O path strongly increase performance. This is further explained in section 2.3 and 3.2.1.2.

3.2.1.2 Cache

Table 2.1 shows the performance of the different parts of the I/O-pipeline. This indicates that increased cache hit-rate increase performance. Better hit prediction by prefetching and larger cache increase hit-rate, see equation 2.4.2. General peformance improvements for the different parts in the I/O pipeline is shown in table 3.1.

	Client		*Server*	
	Prefetching	*Delayed write*	*Server striping*	*Cache*
Network	Lower	Higher	Lower to each	-
Server computer	Lower	Higher	Higher on each	Higher
Client computer	Higher	Higher	-	-

Table 3.1: Performance at high load, client and server cache

GFS uses client cache with locking on the disks while PVFS incorporates server caching. The GFS clients can act as servers for other distributed file systems. This opportunity will not be discussed here as it lengthen the I/O-path and not improve any other attributes than the opportunity to connect with legacy systems. Performance is likely to drop.

Disk cache apply both to GFS and PVFS and will not be elaborated here further than was done in section 2.4.2.1.

As described in section 2.3.3 client cache perform better than server cache. GFS is therefore likely to perform better than PVFS when there are hit in the local cache.

3.2.1.3 Scalability in clusters

The way of exchanging locks on the disks in GFS using DLOCKs does not give fairness and lock blocks of 4KB or 8KB which generate a significantly overhead when the locks have to be exchanged between clients writing to the same data area. Taking one lock is a network access. Thus with 8KB locks locking 1MB gives 125 network accesses before data can be written. The newer distributed lock manager reduces this number of network accesses. This lock manager also avoid clients operating on the same data they to all poll for locks and thereby generate a lot of network traffic. But the clients still have to empty their cache as they release locks. When many clients are doing updates to the same area they will have to drop their cache continuously. This means that the throughput to the clients will drop to the aggregated throughput of the disks. The disks will when accessed by many clients (20-50) have very random access which will decrease performance. This can to some degree be compensated by more striping over more disks. The only problem with this is the number of transactions. The number of disk accesses is still the same even with more disks in parallel. There is a limit to how fast the disk head can move on disks.

The bottleneck in PVFS is the meta-data manager. For small files and many meta data operations performance drops horribly. For large files this is not a problem and PVFS have been shown to produce aggregated throughput higher than 1.5GB/s [Bae02] for the same data. With many clients and many transactions the PVFS server have the advantage of large memory caches. This is not dependent of mechanic parts, each server is able to provide more throughput then each disk and the system scales better.

3.2.1.4 For workstations

Office environments have a lot of small files which is mostly accessed in a continuous access pattern. Small files generate a lot of meta-data operations.

For GFS the performance from local cache is far better than that of network accessed cache. For GFS network accessed cache is by far much faster than that

of disk access. There is almost no simultaneous operations on the same data.

PVFS suffers from the amount of meta-data operations that have to go through the meta data server and the overall performance drops far.

3.2.1.5 For databases, scientific applications

Databases and many scientific applications are HIPS using large data files. For such systems the disk locking scheme and client cache become a problem when scaling while PVFS will perform excellent. Adding more servers to PVFS will increase aggregated throughput.

There is advantages for HIPS to have a seamlessly integration between programming environment and file system when developing parallel applications for clusters. PVFSs support for MPI-IO through ROMIO is therefore beneficial. There is currently no MPI-IO implementation for GFS.

There are however scientific applications that are LCPS. In such a system GFS will perform better than PVFS because the use of local cache instead of cache on the servers.

3.2.1.6 Conclusion

In an office environment GFS works better than PVFS. For use for databases or for many scientific applications PVFS would probably be the better choice except for loose connected parallel applications.

3.2.2 Suitability as a HIPS cluster file system

For a HIPS, performance and scalability is the most important aspects. Table 3.2 shows this for the DFSs described in this chapter. Other attributes as consistency, availability and source availability is listed in table 3.3. The tables show the difference in characteristics of HIPS and LCPS. PVFS is a good file system for HIPS.

3.2.3 Access to source

There have later years been said a lot about open-source and licenses. For this thesis, the impact have been that because the source of PVFS was available, changes could be made. If this had not been so, this thesis would not have had the part that is chapter 4.

Ten years ago, access to the source of parts of parallel systems was more sparse. It can seem that the future will increase availability of source in the more common parts of parallel computing.

3.3 Concluding remarks

As mentioned in the beginning in this chapter. There is no single system or technology that excel in all situations. As of today, AFS, Coda and GFS provides good solutions for more loosely coupled parallel systems. PVFS have proven itself to be a good solution for systems with tight coupled parallel programs.

	HIPS		LCPS	
File system	**Scalability**	**Performance**	**Scalability**	**Performance**
Sun NFS	Bad, server a bottleneck and saturate rapidly.	Bad, writes get lost.	Fair, have to replicate data at the cost of consistency	Good.
Zebra	Good, workload is distributed among servers. But only one cleaner.	Good, file striping and log. file system.	Fair, workload is distributed among servers. But only one cleaner.	Good, file striping and log. file system.
xFS	Good, workload is distributed among servers.	Good, file striping and log. file system.	Good, workload is distributed among servers.	Good, file striping and log. file system.
GFS	Bad, cache coherency between the clients does not scale.	Bad, cooperative cache gets swamped.	Fair, limited by the interconnect to the disks and the cooperative cache.	Excellent, local cache
PVFS	Good, workload is distributed among servers.	Great, more servers increase performance.	Fair, distributed workload.	Bad if small files. Good if large files.
OpenAFS	Bad, client cache does not scale and too coarse locking.	Bad, cooperative cache gets swamped.	Excellent, distributed workload	Excellent, local cache.
Coda	Bad, can not do simultaneous isolated changes in files.	Bad, cooperative cache gets swamped.	Excellent, distributed workload	Good, local cache, but large operation granularity.

Table 3.2: DFS performance and scalability

File system	Consistency	Availability	Source availability
Sun NFS	Poor. Just for read-only directories.	Poor. No handling of server downtime.	Compliant open-source versions exists.
Zebra	Excellent. Unix semantics.	Good. Tolerates up to one disk or server crash in each stripe.	Research system, stopped development.
xFS	Excellent. Unix semantics.	Good. Tolerates up to one disk or server crash in each stripe.	Research system, stopped development.
GFS	Excellent. Unix semantics.	Poor. Recovery needs manual handling.	Was open-source, became proprietary.
PVFS	Good. Have checkpoint functionality.	Poor. Optimized for speed not availability.	Open source, licensed under GPL.
OpenAFS			
Coda	Good, coarse granularity.	Excellent, supports disconnected operation.	Open source, licensed under GPL.

Table 3.3: DFS attributes

Architecture	Performance	Fault handling	Scalability
One server	Good	Single point of failure, ok recovery	Bad - low performance increase
Split across servers by directories	Good - as with one server	Other parts still work, ok recovery	Very good, but with hotspots - bad, as with one server
Striped servers	Good - as with one server, higher aggregated	May tolerate server failure, ok recovery	Good - handles hotspots but need synchronization
Three level	Good - as with one server	May tolerate server failure, ok recovery	Very good for read-only hot spots, for read-write hot spots - very bad

Table 3.4: Architecture comparison

In LCPSs, the ability to cache locally is the main technology to increase performance. In HIPSs, the ability to distribute load have proven the most beneficial. Most LCPSs perform bad as HIPSs and vice versa. For LCPSs this is because local cache create cache coherence problems with tight integrated parallel programs. HIPSs might be fair as LCPSs, but scalability is worse.

As shown in table 3.4 each architecture have its uses. In the case of the only shared media is the storage system, communicating directly to the server as in *one server* or *many servers* is the most responsive way to communicate. If scalability is an issue, striping servers solves this best. Three level architecture makes it possible to maintain existing infrastructure while benefiting from newer technology.

Chapter 4

PVFS over SCI

As mentioned in chapter 1 the environment in focus for this thesis uses cluster, MPI and SCI for high performance computing. PVFS supports MPI and have good performance results in cluster environments. In chapter 2 it was stated that a distributed file system can never be as fast as a local because of the lower through of the network. SCI have fundamentally more throughput than Ethernet and should therefore improve performance.

This chapter covers the design and implementation of a protocol for SCI and the change to this SCI protocol in PVFS. This have increased performance of PVFS compared to 100Mb Ethernet by a factor of minimum 5 for the SCI cards that was used which had a throughput of 544Mb/s. The PVFS aggregated throughput follows closely the throughput of the interconnect hardware and it is expected that existing SCI cards with a throughput of 315MB/s (2520Mb/s) will improve performance even further.

4.1 PVFS

There are many reasons why PVFS were chosen for enhancements in this thesis. Some of this are: PVFS is used in the same environments as SCI, high performance clusters. It is supported by ROMIO which in turn support MPI-IO. The low latency of SCI was considered a potential speedup for meta-data operations in PVFS. The technology used in PVFS have proved scalable with high performance. A PVFS is clients, multiple I/O servers and one manager. Metadata operations is handled by the manager.

The source code of PVFS is easy to get the hand on since it is licensed under the GPL. There are also actively development being done to PVFS with support mailing lists and more.

While PVFS is based on TCP/IP over Ethernet, there have also been written an implementation for Myrinet [CLRT00]. SCI and Myrinet both are low latency high throughput interconnects. PVFS have during this thesis been adapted to SCI. The test results is given in this chapter. PVFS is described in more detail in section 3.1.5 and 3.2.1.

	Throughput	Latency
TCP/IP 100Mb	12 MB/s	97us
SCI interconnect	69.8MB/s	7-1.4us

Table 4.1: Throughput and latency in TCP/IP and SCI

4.2 SCI

The Scalable Coherent Interface (SCI) was introduced as a draft [SCI91] in 1991 and became and IEEE standard [SCI93] in 1993. It has also become an ANSI standard for multiprocessors, specifying a topology-independent network and a cache-coherence protocol.

SCI hardware provides a low latency, high throughput interconnect for shared memory between different machines and implemented in standard PCI cards. SCI is mostly used in cluster environments because of the high throughput, low latency and relative short cable length. Dependent on SCI cards and PCI hardware throughput have been measured to 60-326MB/s and latency to 7-1.4us.

The hardware used during this thesis use routing in the cards themselves and thus no central switch. They are connected to each other using rings where each have a next and previous node. This scale by using more rings. One ring is the simplest way of connecting machines or nodes as is a more used expression. For further scalability nodes may be connected in a mesh where each card also do routing between two rings. Next step in scalability is setting it up as a cube and then again as a hypercube. This have proven very scalable. In the system used for the tests transfer from one ring to another was measured to 3us.

In this thesis SCI is only used as a high throughput low latency interconnect. The cache coherence protocol is not used. Table 4.1 shows a numeric comparison of TCP/IP over Ethernet and the SCI interconnect. Myrinet as described in [BCF+95] have lower throughput and higher latency than SCI.

4.2.1 SCI API

The SCI drivers and support software used is copyrighted and made by SCALI, the SCI hardware is made by Dolphin Interconnect Solutions. The API layer used is minimalistic and efficient. It consist of four functions:

openLocalChunk()	openRemoteChunk()
closeLocalChunk()	closeRemoteChunk()

To start communicating with SCI one only have to open a local memory area with openLocalChunk() and then let the remote node connect to this area with openRemoteChunk(). After this the remote node may write directly to the local memory.

4.2.2 Checkpoints

Even as the SCI interconnect is a point to point interface, data might get lost. This happen if there is too much data received by a SCI for the destination to handle. This destination might be the PCI bus in the computer or an outgoing link on the SCI card. The later is possible since there might be more than two connection on a SCI card.

The software drivers for the cards that were used had a checkpoint function that flushed the data onto the interconnect and checked for errors. It is thus easy to achieve TCP like functionality by setting up a loop around the memory copy function. If there is an error the memory copy should be done again.

4.2.2.1 Boundaries

Depending on the version of the card there is a boundary for every 64 or 128 byte. When data is written to the boundary the data will be flushed. This means that when the checkpoint function is called, the data is already sent and the function will just return with the status. This will decrease latency when sending small messages.

4.3 SCI protocol

To replace the network backend of PVFS, a new protocol was designed and implemented with similar attributes as TCP/IP. This is described in this section. The hardware used for the tests in this thesis support remote write but not remote read. This had impact to the design of the SCI protocol.

4.3.1 Architecture

There are differences between communication through streams as in sockets and shared memory that is worth noticing. First of all there is no automatic send or receive buffering. I have written a protocol based upon ring-buffers to get this. Second therefore there is no way that two connecting computers will not overwrite and corrupt each others data if using the same memory area. To handle this I have set up a connection area on each host where each host have its own exclusive area to write connection data to. When connections are set up, areas are exclusively allocated.

4.3.1.1 Connection area

Each node shares a connection area to all the others. On this area there is one subarea for each node in the cluster. This is done because shared memory does not give any queuing and therefore needs exclusive access on areas. This way, a node may send information about connection and disconnection to other hosts without the risk of non-consistent data. But only as long it does not write to the area again itself too soon before the receiving computer gets to read it. I have therefore added a timer to avoid this. *Change indicator* is used to indicate new data on the connection area. *Service* indicate the service connected. There are also some SCI specific fields that specify the local communication area for the connected host to connect to. These fields are *module id*, *chunk id* and *chunk size*.

The connection id field identifies the connection at each end. It is used at connection and disconnection. It has two parts. The upper 16 bits and the lower which on receiving are remote and local connection id respectively.

4.3.1.2 Communication buffers

The communication buffers consist of a buffer area, the front pointer for the local buffer and the end pointer for the remote buffer. The reason for this mixing of local and remote pointers is that the SCI implementation used only support remote write, not remote read. Therefore, the pointers that a host updates have to be written to the remote communication buffer.

The buffer area is used as a ring-buffer and the pointers is used as the front and end pointer. The front pointer is pointing to the address after the last data written. The end pointer is pointing to the address after the last data read. If front pointer equals end pointer, the buffer is empty. None of the pointer may pass the other by advancing.

4.3.1.3 Link identifiers

The communication link identifiers that is used throughout PVFS is socket numbers. Therefore the identification of communication links and listening ports is numbers in the SCI protocol to ease the exchange of protocols.

It is this link identifiers that are transferred using the connection-id field in the connect area.

4.3.2 Procedures of usage

4.3.2.1 Initialization

Each node have to initialize the SCI libraries and offer the connection areas for connection. This is somewhat similar to binding of sockets but is not specific for special services.

4.3.2.2 Listen for connection or data

Connections and disconnections are noticed by polling for change in the local connection areas change indicators. Similarly data is noticed by comparing the front pointer to the end pointer.

4.3.2.3 Connection

Here I assume the API is already initialized. A connection goes like this:

1. The connecting host allocates and offers a receiving buffer.
2. The connecting host writes the module id, chunk id, chunk size, service and connection id for the offered buffer area. ensure they are written ok, write the change-id and ensure that it is written ok.
3. The connected host detects the change in change-id and reads the connection field. The local connection id is not given and the host connected understands that it is a connection.
4. The connected host allocates and offers a receiving buffer, connects to the remote receiving buffer indicated by the connection field and makes a connection to the connecting node in the same fashion as described above with references to the local receiving buffer. The connection id is now set with both the local and the received connection id.

sci_initiate_connect()	ensure_initialized()
sci_finish_connect()	sci_handle_connect()
get_request()	sci_disconnect()
get_node_request()	is_new_data()
sci_send()	is_new_connection()
sci_recv()	

Table 4.2: SCI protocol API

5. The connecting host detects the change in change-id and reads the remote connection-id.

Connection established.

4.3.2.4 Sending data

Sender checks if room in the recipients receiving buffer by examining the front and end pointers. If it is it will write the data. Then ensure that the data is written ok, updates the front pointer and ensure that it is updated ok. This implicates that the front pointer newer is moved before data is written ok. Consistency is thus maintained.

4.3.2.5 Receiving data

Receiver checks if there are data in the local receiving buffer by examining the front and end pointers, and if it is, read the data. It then moves the end pointer and ensure it is written ok to the remote host. This order ensures data is not overwritten by the sender before the receiver reads it.

4.3.2.6 Disconnecting

A disconnection goes like this:

1. The disconnecting host writes the connection id with the remotes local connection id. Then the change indicator is changed.

2. The receiving host of the disconnection reads the disconnection request. Closes down the connection and writes back a disconnection for the same connection to the initiating host.

3. The disconnecting host reads the request and closes down the connection.

Handling of the case that the receiver of the disconnection does not answer is simply a timeout and the initiating host continues. The initiating host can not close down the connection immediately after the request is sent since the recipient of the request might try to send some data in the time between the closedown and the handling of the request for disconnection. This will cause an unwanted shared memory error.

IOD		MGR	
	IOD		
		IOD	
			IOD

Figure 4.1: Placement of the I/O nodes in the mesh

4.3.3 Protocol API

The protocol API is given in table 4.2. The reason for splitting connection into two functions were that the protocol were developed on a uniprocessor computer with shared memory. This splitting enabled the test programs to handle both ends of the shared memory communication links. It should be noted that this were only for testing purposes. The time between time slices in the kernel scheduler is to long on a uniprocessor to play both ends in the communication. A simple ping-pong test using shared memory shows that it takes $>$ 300ms to transfer control from one process to another (using Linux kernel 2.4.19) and thus rendering local low latency communication impossible and high throughput impossible.

4.3.4 Overloading the interconnect

With 4 or more simultaneously senders to the same node performance dropped to below the tenth of the max capacity of the receiving node because of write errors and frequent retransmissions. This happens even as the SCI interconnect is a point to point interface because the routing and resending of data in the SCI card have no means to store and send data later. The reason for this become apparent when studying the mesh layout of the SCI interconnect. As data is routed from one node to another, one SCI card might receive from several SCI interconnect rings what is to be routed onto the same SCI ring. The same happen when a computer node receive data from several SCI rings at the same time. This is not a simple matter of chips in the cards as neither the system bus in the computer can handle such amounts of data in the tempo they arrive.

4.3.4.1 Placement of the PVFS server nodes in the mesh

A PVFS system is one manger, a number of I/O nodes and clients that uses the system. To fully utilize the system, the number of clients are larger than that of the I/O nodes. Each of the clients will communicate with all the I/O nodes at the same time. The I/O nodes will under heavy use communicate with all the clients at the same time. Since there are more clients than I/O nodes the I/O nodes will get the largest impact of performance.

To avoid overloading the interconnects, these nodes should share as few rings as possible. This is typically done by placing the nodes diagonally in the mesh like in figure 4.1. The routing in the SCI mesh is not dynamic load balanced.

4.3.4.2 Problem consequences

With ten clients all writing to the same servers some data just did not manage to get through even as the timeout of given up was set as high as 10 seconds.

This was with a 2 dimensional mesh grid of 16 computers. It should be noted that this was not the newest SCI cards.

4.3.4.3 Exponential backoff

There were set an upper cap of 1024us to the backoff. The delay was applied when write error occurred. With this applied, the problem described above was reduced to lowering the utilization of the interconnect.

The backoff has a random part to avoid repeatedly collisions. The glibc function rand() is slow to generate a random number. To increase the speed of getting random numbers, some thousand numbers were generated with rand() and stored in an array during initialization and fetched one at a time when needed in a round-robin scheme.

4.3.4.4 Token based write access

Another approach tried was to give each node a number of write tokens equal to the maximum number of writers the node can handle. When combined with a diagonal placement of I/O nodes this should improve scalability as long as the time to transfer a token is not too long.

4.3.5 Small writes and latency

The receiver can not start reading data before the first record is finished written in the receive buffer. When the sender is finished sending, the receiver still have to receive the last record. This means that the total time from send starts to receive is complete is shorter when the record size is smaller. If there is an error when writing data, small writes means a smaller set-back. Small writes have been implemented in the SCI protocol by a size of 64KB.

4.3.6 Flushing boundaries

Section 4.2.2.1 explained how the flushing boundaries in SCI work. There were evaluated if this could benefit the performance of PVFS or generally improve the protocol. When a computer used have a 1GHz CPU, 1000 instructions is done in a micro second. The CPU therefore have time to do some extra work to help speed things up.

Chapter 2 stated the need for buffers in a data pipeline. If a simple ring-buffer were used and its end were aligned on a boundary then the only place where extra alignment would be needed for extra speedup would be at the end of a write. If each end were end-aligned on a boundary data would be sendt faster to the recipient. This were implemented by writing the data between the boundaries and writing a number on the boundaries to indicate how much data were present between the latest boundaries. Testing of this showed no improvement in performance for very small writes (1KB). The network buffers are large because the recipient might not be ready to receive the data yet. This leads to the conclusion that saving a couple of micro seconds on the last write is fruitless and can thereby be left as it is. The final implementation of the protocol did not use this boundary alignment except for the last part of the buffer.

```
SCI_WRITE_START {
        memcpy( &(con->r_buffer[con->r_frontp]), buf,
                small_write_size );
} SCI_WRITE_END;

data_left -= small_write_size;
con->r_frontp += small_write_size;

SCI_WRITE_START {
        *(con->r_frontp_s) = con->r_frontp;
} SCI_WRITE_END;
buf += small_write_size;
```

Figure 4.2: Use of the START and END macroes

	Throughput	Latency
TCP/IP 100Mb	12 MB/s	97us
SCI protocol	65MB/s	17-20us

Table 4.3: Throughput and latency in TCP/IP and the SCI protocol

4.3.7 Constructive use of macros

In section 4.2.2 there were described how a loop construct around a memory copy operation could be repeated to achieve TCP like functionality. There were chosen to make two macros to handle this loop and exponential backoff. The use is showed in figure 4.2 from a part of the sci_send function. The macros simplifies the code and work independent of how the shared memory is written. In the figure both a memcpy and an assignment is used to update remote memory. The memcpy function is used because it at the actual installation had equal performance to the best of the specialized methods like using MMX.

Figure 4.3 show how the macros were implemented. The first and last part of the loop is in the START and END macro respectively. The exponential backoff is seen as the variable *backoff* where the maximum backoff is 0x7ff - 1ms. A "@" is printed for every retransmission and if it is not possible to transfer the data within *SCIWRTOUT* micro seconds a "£" is printed out to indicate that there is something wrong. The FLUSH_CPU statement is another simple macro to flush the CPU cache to ensure that the content have been written to memory.

4.3.8 Performance

Table 4.3 shows a numeric comparison of TCP/IP over Ethernet and the SCI protocol implemented during this thesis. The SCI cards used had a 70MB/s throughput and the SCI protocol thereby had a 7% overhead. The throughput and latency of TCP/IP were measured on 100Mb/s Ethernet cards in the same computers as the SCI cards. There is no part of the protocol that should limit the performance from scaling with the SCI hardware. Buffer sizes can easily be increased and 326MB/s cards should be able to push data at 300MB/s with the protocol.

```
#define SCIWRTOUT   10000000LL // 10 seconds


#define SCI_WRITE_START \
{ \
        unsigned errorcntr; \
        long long start_time, end_time; \
        long long backoff = 16; \
        BOOL ret; \
        EX_LATENCY; \
        errorcntr = SciSampleErrorCounter( 0 ); \
        start_time = get_time(); \
        end_time = start_time + SCIWRTOUT; \
        do { \

#define SCI_WRITE_END \
                FLUSH_CPU; \
                if ((ret = SciCheckpoint( 0, &errorcntr )) == FALSE ) { \
                        printf( "@"); \
                        backoff = ((backoff << 1) | 1) + \
                                (get_fast_random_number() & 0xf) & 0x7ff; \
                        if ( get_time() < end_time ) delay_us(backoff); \
                                continue; \
                } \
        } while (0);
        if ( ret == FALSE ) \
                printf( "£" ); \
}
```

Figure 4.3: The SCI_WRITE_START and SCI_WRITE_END macros

Wrapper function	TCP/IP	SCI protocol protocol
sci_connect()	connect()	sci_initiate_connect() sci_finish_connect()
sci_bind()	bind()	
sci_read()	read()	sci_recv()
sci_write()	write()	sci_send()
sci_send()	send()	sci_send()
sci_poll()	poll()	is_new_data() is_new_connection()
sci_select()	select()	is_new_data() is_new_connection()
sci_close()	close()	sci_disconnect()
sci_recv()	recv()	sci_recv()
sci_accept()	accept()	sci_handl_connect()

Table 4.4: Wrapper functions

4.4 Implementation in PVFS

In PVFS their is a separate socket communication API. This would make a transaction to another protocol very easy. Unfortunately this is not exclusively used as there are multiple examples of manipulation of the sockets directly through libc. Because of this there was done a reimplementation's of the isolated communication functions in addition to wrapper functions for the socket functions listed in table 4.4. Then the wrapper functions had to be called instead of their TCP/IP counterparts.

Since PVFS not yet have a separate network API that may be changed seamlessly, the changes to the PVFS code base was made as small as possible to be more easily maintained across versions. The result was easy implemented and should also be possible to use to replace TCP/IP in other applications.

Some of the communication in PVFS is separated into an API but this is not sufficient to replace the network backbone. The implementation uses the protocol described in section 4.3 with an extra layer of functions to simulate communication through sockets. This functions had identical signatures to that of the functions in the socket API.

4.4.1 Splitting and adaptation layer

Adaptation and splitting communication between applications written to use TCP/IP and the SCI protocol. For short this will from here just be called adaptation layer.

4.4.1.1 Wrapper functions

To achieve this there were written wrapper functions of those socket functions that was used in the PVFS code. These wrapper functions then switched function calls to the original functions or the appropriate functions in the SCI protocol were appropriate. Table 4.4 lists these functions and show how they split control between TCP/IP and the SCI protocol.

Figure 4.4 shows the wrapper function for read. Note the print_info() function calls. These are parts of the debugging system that I wrote to get it all to work. The rest quite small and straight forward so replacing the SCI protocol with other protocols with similar API should be quite effortless.

Table 4.5 show how the wrapper functions is placed as a splitting and adaptation layer between the PVFS internal semantic and the respective

```
ssize_t sci_read( int filedes, void *buffer, size_t size )
{

        print_info( "SCI_READ" );
        ensure_initialized();
        if ( get_map_socket_con_idx( filedes ) != -1 ) {
                print_info( "reading with sci" );
                return sci_recv( filedes, buffer, size, FALSE );
        }
        print_info( "reading without sci" );
        return read( filedes, buffer, size );
}
```

Figure 4.4: Wrapper function for read()

PVFS	
Splitting and adaptation	
TCP/IP	SCI protocol
Ethernet	SCI

Table 4.5: PVFS network layering

protocol APIs. This architecture makes it simple to use SCI for some services and TCP/IP for others. It is the IF statement in figure 4.4 that does the splitting.

4.4.1.2 Simple mapping

Mapping between numbers is used several places in the adaptation layer. Socket numbers are mapped to array indexes for the SCI connection information, services associated with the listening sockets, listening sockets associated with the services, association from listening socket to node_id and a simple queue for the connection requests, mapped from socket number to array index for SCI connection information.

This were implemented in a small library that provided such mappings as hash functions and operations on them.

4.4.1.3 Initialization

In figure 4.4, ensure_initialized() is is called at line two in the function. It is a part of the wrapper function layer to ensure that the protocol and middle layer is initialized. The first part in this function is a test to return immediately if initialization have already been done. Since function calls are so cheap (short time usage) in the C programming language I have included this function calls in every wrapper function and thereby ensuring that initialization without the application using them having to change.

4.4.1.4 The select function

The select function in standard *libc* is at the heart of communication. It detects connections and data and pass this knowledge on as socket numbers in a bit array. The SCI protocol have means to detect both connections and data and the select wrapper function could thereby use these. Since the SCI connections are designed to be identified by numbers there was sufficient to reserve a socket number for each SCI connection and pass this on. There was then made hash mappings between the different numbers for fast lookup.

The standard select function have a timeout that allows passive waiting. The SCI libraries does only partially support such passive waiting and this was not used in the implementation. Thus polling were used. Because of the nature of select, being able to monitor several sockets and file descriptors at once, the wrapper function for select had to handle both ordinary descriptors and SCI sockets fairly. Thus the wrapper function ended up both calling the ordinary select function on the bit arrays minus the SCI sockets and polling on each SCI socket present in the bit arrays. There are thus no passive waiting except for the one done in the original select function and this have to be set very short to not add to much latency for the SCI part. The impact for this on programs running on the same host have not been measured. But because of the results in section 4.5 some latency should not decrease performance too much.

4.4.1.5 The sci_hosts file

An example of a sci_hosts configuration file is shown in figure 4.5. It has a generic format suggested by one of the PVFS developers. This file is parsed at initialization and serves several purposes:

Mapping IP-address to protocol support is done by the protocol indication first on the line ("sci://"). This is done on suggestion by one of the PVFS developers as this might be the future format for a hosts file for PVFS when using multiple protocols.

Protocol choice by service To test how different services benefited by SCI, the design was made to only enable use of SCI for services listed in this file. This meant that real sockets was used as communication identifiers as described in section 4.3.1.3 and SCI was only used for communication when the connected service indicated it. When a socket was used to connect to a service that was supported by SCI, the connection and communication was done by SCI while the socket was associated to the SCI communication channel in the adaptation layer. Likewise, if a socket is set to listen on a port that is supported by SCI, the adaptation layer register the port associates the socket to it.

For each of the servers listed the services 3000 for the meta manager and 7000 for the I/O daemons is supported by SCI. The node_id on the same line as the services identifies the SCI NIC of the host contacted. Both the node_id and service is given at connect and is then matched with the info in this file. This file should be the same for all files on all nodes.

Choice of IP-addresses for SCI entries can be any IP-address but one should avoid conflicts with used addresses. The point is to give the application

```
# The format of the line:
# protocol://IP-address: parameter1, parameter2, .., parameterN
#
# For sci the line is as follow:
# sci://IP-address: node_id, module_id, chunk_id, service1, service2, ..serviceN

sci://127.0.0.2:    0x0100,   1000, 5551,     3000, 7000
sci://127.0.0.3:    0x0200,   1000, 5552,     3000, 7000
sci://127.0.0.4:    0x0300,   1000, 5553,     3000, 7000
sci://127.0.0.5:    0x0400,   1000, 5554,     3000, 7000
```

Figure 4.5: The top of the sci_hosts configuration file

an ordinary IP-address that it can handle as is will but when connecting it act as a mapping to the SCI connection data when going through the adaptation layer. In the example in figure 4.5 there are used local addresses instead of the real ones. This were done because the tests were run on different clusters and the node_id were often the same but the IP was of course not.

Connect info is also given in the sci_hosts file. These are 1000 and 5551 for the first entry and is the module_id and chunk_id for the connection area for the specific host. One host have only one connection area that is shared to all the other computers. Each node have its own section of this connection area as described in section 4.3.1.1. The place within the connection that is used for a specific host is calculated based on the order the host have in the sci_hosts file and the size of each connection section.

Adding more protocols is the whole idea of the "protocol://" format. The only changes needed would be to change the *if* statements in the wrapper functions to *switch* statements.

4.4.2 Debugging

During the debugging of a program it improves efficiency greatly to have a good overview of what the programs does as they run. In single process programs the usual solution is to print debug messages at different points in the program. The same approach can be used in distributed systems but less efficiently. The problem is that each process prints its own messages and there is no ordering between processes.

The solution I choose for this was four fold. Firstly, to order the debug lines from different processes, at the beginning of every debug line there were printed the present time when the message where printed. Thus all the debug lines could be sorted in right order. Second, to separate debug lines from different processes, the process name was printed after the current time. Third, to reduce the programmers extra work of the first and second part, this was packed into some macros. This also gave the opportunity to easily print the file name and line number of the debug statement. Fourth, since distributed applications are often big applications and the number of debug statements immense there was

added a one place handling of which debug statements to be run. This control was divided into which part of the code and the type of messages.

Figure 4.6 shows a sample of this debug output. The process names are visible as *test_server* and *test_client*. They are automatically fetched from the */proc* virtual file system during initialization. The types of message visible is *INFO*, *DEBUG* and *POINTERS*. File names and line numbers are given within the set of parenthesis. The function name where the debug statement occurred is next before the actual debug message.

If an error occur, switching on more verbose debugging gives a very fast indication of where the error occurred *and* what the rest of the distributed application were doing at the moment.

4.4.3 Protocol window size

The protocol window is the receive buffer. It should be at least 256KB in the SCI protocol to not limit performance when used with PVFS. During the tests a receive buffer of 2MB was used. It is expected that such sizes also matter for other protocols used with PVFS.

4.4.4 Testing procedures

To catch possible breaks in the code, tests for the different functionality were implemented to be run from the Makefile with *make tests*. Since these tests were dependent upon the executable, it was a on-line compile and test that helped development.

Because of the frequent running and great number of tests there were chosen to limit the running of each test by time and not by number of repeated operation. How long one run took were often unknown and this way the tests run not too long while they still had ok results.

For the test results in this thesis, the tests were run for a longer time and number of repeats were controlled to ensure reasonable accuracy.

4.4.4.1 Start and stop in aggregated tests

The nodes does not start nor stop the test at the exact same time. Measured aggregated throughput will therefore be too high since there will be more resources available when not all nodes are active at the same time.

To handle this, there were written a simple test program that in addition to running the tests also ran the same tests for a little while before and after the actual testing. This ensured that no node at any time during the tests had more resources available.

4.5 Network latency impact on PVFS

An application that is waiting for data is not utilized and the overall performance of the application decrease. Network latency is therefore important since applications communicate using networks. How much waiting which is done is difficult to measure. This is because the intervals often are very short and to time them would actually increase the latency. Also, the overall performance is what's important.

```
1036072527 779184 test_server  INFO: (sci_protocol.c, 293) in sci_send: SCI_SEND
1036072527 779190 test_server  INFO: (sci_protocol.c, 295) in sci_send: SOCKET = 7, LEN = 131072
1036072527 779197 test_server  INFO: (mapping.c, 98) in get_mapping: GET_MAPPING
1036072527 779203 test_server  DEBUG: (mapping.c, 99) in get_mapping: map at: 0x8075d10, from: 7
1036072527 779210 test_server  DEBUG: (mapping.c, 100) in get_mapping: map->numOfMappings = 256
1036072527 779217 test_server  DEBUG: (mapping.c, 110) in get_mapping: LEAVE FOUND: 0
1036072527 779223 test_server  DEBUG: (sci_protocol.c, 299) in sci_send: con_idx = 0
1036072527 779230 test_server  POINTERS: (sci_protocol.c, 300) in sci_send: START l_fp: 0xd5 l_ep: 0xd5 r_fp: 0xe00d5 r_ep: 0xd5
1036072527 779266 test_server  POINTERS: (sci_protocol.c, 347) in sci_send: write 128K 0 bytes
1036072527 779275 test_server  POINTERS: (sci_protocol.c, 349) in sci_send: write address: 0x4045f0dd
1036072527 809302 test_server  DEBUG: (sci_protocol.c, 379) in sci_send: con_idx = 0
1036072527 809318 test_server  POINTERS: (sci_protocol.c, 381) in sci_send: END l_fp: 0xd5 l_ep: 0xd5 r_fp: 0x1000d5 r_ep: 0xd5
1036072527 809329 test_server  INFO: (sci_protocol.c, 407) in sci_send: LEAVING, 131072
1036072529 589713 test_client  INFO: (sockio.c, 231) in nbrecv: NBRECV
1036072529 589725 test_client  INFO: (sci_protocol.c, 1482) in get_map_socket_con_idx: GET_MAP_SOCKET_CON_IDX
1036072529 589732 test_client  INFO: (mapping.c, 98) in get_mapping: GET_MAPPING
1036072529 589739 test_client  DEBUG: (mapping.c, 99) in get_mapping: map at: 0x8075b70, from: 6
1036072529 589747 test_client  DEBUG: (mapping.c, 100) in get_mapping: map->numOfMappings = 256
1036072529 589754 test_client  DEBUG: (mapping.c, 110) in get_mapping: LEAVE FOUND: 0
1036072529 589761 test_client  INFO: (sci_protocol.c, 428) in sci_recv: SCI_RECV
1036072529 589767 test_client  DEBUG: (sci_protocol.c, 429) in sci_recv: SOCKET = 6, LEN = 131072
1036072529 589774 test_client  INFO: (mapping.c, 98) in get_mapping: GET_MAPPING
1036072529 589780 test_client  DEBUG: (mapping.c, 99) in get_mapping: map at: 0x8075b70, from: 6
1036072529 589786 test_client  DEBUG: (mapping.c, 100) in get_mapping: map->numOfMappings = 256
1036072529 589793 test_client  DEBUG: (mapping.c, 110) in get_mapping: LEAVE FOUND: 0
1036072529 589799 test_client  INFO: (sci_protocol.c, 437) in sci_recv: non blocking
1036072529 589805 test_client  POINTERS: (sci_protocol.c, 439) in sci_recv: START  r_endp: 0xd5,
1036072529 589813 test_client  POINTERS: (sci_protocol.c, 482) in sci_recv: read address: 0x4017f0dd
1036072529 590490 test_client  POINTERS: (sci_protocol.c, 485) in sci_recv: read 131072 bytes
```

Figure 4.6: Sample debug output

4.5.1 Decreased network latency by 80%

To test the overall performance with different latencies I have used the SCI protocol with the adaptation layer with PVFS. SCI have very low latency and is thus suitable as a reference measurement when compared with TCP/IP. The protocol I implemented for SCI have on the hardware used 17us latency from send from application until receive in application. TCP/IP have 97us on the 100Mb/s cards used. Both latency's are measured as the average of 10K ping-pongs /2 when sending 4 bytes.

Test of the protocol in it self showed as expected no change in throughput. This is because of pipelining as described in section 2.3.

For testing the performance change in PVFS there were used one client and two servers. A 10MB file size were chosen to fully cache the file in the servers memory. Record size were 4KB. In the first test the SCI protocol were used with 80us extra latency on sending to compare to that of TCP/IP. This were implemented in the wrapper functions and as active waiting to ensure accuracy. The second test were run without this extra latency.

The results were that there were not possible to distinguish the throughput of the two tests. Each tests were random and continuous, read and write in combinations to a total of 4 test cases.

4.6 Performance results

Because of limited time and availability of testing equipment this results is not as complete as should have been.

When testing performance in PVFS using SCI the overloading of the SCI interconnect became an obstacle. Two nodes writing to one worked fine. Three nodes writing to one were generating collisions and four nodes writing to one generated so many collisions that it had lower aggregated throughput than when using three nodes.

4.6.1 Controlling the senders

To limit this overloading and thereby increase performance different suggestions were discussed. There are two additional problems to this. The SCI communication done by PVFS is just part or the total amount of traffic on a cluster in production. Total control is therefore impossible. The second problem is that PVFS does write and read operations in parallel. This generate a lot traffic that is local in time and therefore prune to overloading.

The best solution to this were found to limit the amount of simultaneous writers to a single node. This would not help with collisions on the network before it reached the node, so a simple exponential backoff were implemented similar to that of IP.

First a *grant-write* scheme were studied. The writer would have to be given permission to write the amount requested. Because of the amount of requests needed even when there were no problem with overloading, this were rejected for another approach. This used *write-tokens* for the right to write. Each node have a number of write tokens. This number should be the number of simultaneous writes it can receive without problem. When other nodes want to write (send) to the node they will first have to query for a token via the connection area.

Nodes			
1 - 1	Latency	16.58us	(no change)
1 - 1	Sending	65.3MB/s	(no change)
2 - 1	Sending	98MB/s aggregated with 20 tokens	(no change)
2 - 1	Sending	70MB/s aggregated with 1 token	(30% perf. drop)

Table 4.6: Token impact on protocol performance

	Exponential backoff	Tokens
Continuous read	47.46MB/s	29.88MB/s
Continuous write	29.95MB/s	52.42MB/s
Random read	48.46MB/s	52.30MB/s
Random write	29.88MB/s	29.95MB/s

Table 4.7: PVFS over SCI using 1 client and 2 I/O nodes

The writing node keeps the token until the node wants it back. This means that the overhead when no limiting of write is needed is theoretical zero. When all the tokens are given away, the receiving node will upon further requests query to get the token back from the node that have had its token the longest. This ensure fairness.

The impact upon introducing tokens to the protocol is shown in table 4.6. The problem here is the big performance drop of 30% when switching tokens back and forth. One token request / give token is 2× sci_checkpoint (5us) and 2× transfer (5us). The whole process includes one request for token, one request to get back token, one request to give token back and one request to give token. Totally 4 requests of 20us. This should theoretically give 80us but tests have showed that taking a token back from one node and giving it to another take about 220us. Since the CPU at 1GHZ can do about 1000 instructions in 1us. The very small code pieces to handle the tokens (about 20 lines of code in C) should not be the reason for the extra latency and other reasons have not been found.

The most obvious way to increase throughput when using tokens would be to either reduce the latency for the token exchange or let the writers hold the tokens for a longer time. The latter would have been implemented if there had been enough time.

4.6.1.1 PVFS and the token implementation

The tokens work when testing the SCI protocol and in small test applications. They do however not work in the PVFS implementation if the nodes dont have tokens equal to the number of connections. This might be a timing problem in PVFS or the protocol but it has not been further probed within the time available. Table 4.7 show performance results when using only exponential backoff and combined with tokens. To make this test run, the number of tokens were 3. The reason for lower throughput on some values is unknown but it is not due to collisions and retransmissions as this were low to non existing during the test.

The file size was 10MB to let the servers cache the whole file and the record size was 1MB. The record size where chosen because PVFS over TCP/IP and

	Throughput
Continuous read	11.71MB/s
Continuous write	11.71MB/s
Random read	11.70MB/s
Random write	11.71MB/s

Table 4.8: PVFS over Ethernet using 1 client and 2 I/O nodes

SCI have shown higher performance for higher record size. This is half the window size used during the tests.

The performance for PVFS when using TCP/IP over 100Mb/s Ethernet is shown in table 4.8. A file size of 10MB and a record size of 1MB was used as in the test with SCI. The protocol window size where set to 1MB in the kernel during the test.

Chapter 5

Conclusion

In search for a high performance cluster file system several file systems have been evaluated and PVFS has been chosen as one of the best and enhanced with a higher throughput network.

Building a simplified protocol with features resembling TCP/IP using the shared memory of SCI have proven feasible and efficient. The protocol have shown a 7% overhead. Multicast and some other more exotic elements of TCP/IP communication is not implemented. An adaptation layer with wrapper functions for the most used socket functions have proven it easy to replace TCP/IP with SCI in generic applications. The adaptation layer also have the ability to use different protocols and networks for different hosts and services.

PVFS have about 5× higher throughput with 70MB/s SCI cards than 100Mb/s Ethernet. With higher throughput SCI cards there is expected a further increase in performance. There exists SCI cards with a capacity of 326MB/s sustained throughput.

The SCI protocol implemented here have 80% less latency that TCP/IP over Ethernet. The lower network latency of SCI have not shown any gain in performance due to extensive use of pipelining.

Overloading the interconnect was a problem with the protocol implementation. Exponential backoff have improved this but token based reservation of capacity should improve this further.

Buffering is important to increase parallel execution and thereby throughput. Too small buffers lowers throughput while very large buffers is waste. Buffers in a distributed file system is on-disk buffer cache, network buffers and application buffers.

High bandwidth networks and local I/O Because of the faster increase in network throughput than for the internal bus, the local I/O path have become an issue for distributed file systems. Higher network throughput means the network buffers is filled faster and should be larger.

Disks can have the throughput increased with RAID, caching and log-structuring. Record size have shown to have a performance impact of about

40% on local file systems because of the block size in the OS and CPU cache. It is expected to have similar impact on distributed file systems.

Simultaneous updating same data from clients is only supported by *parallel* file systems. PVFS handles this well by striping the data onto the servers to spread load and using cache implemented on the servers to increase performance and scalability. A shared disk approach with similar striping is evaluated and found to not scale equally well because of the lack of server cache.

Client or server cache in a cluster The amount of memory available for caching disk data is crucial for disk performance. Both because it increase throughput and because it avoids disk access. Disks are mechanical devices that can respond to limited number of requests per seconds.

Local (client) memory used as disk cache gives obviously a throughput equal to that of memory when cache hit. Remote memory used as disk cache gives a throughput equal to the minimum of the network and the PCI bus which is typically $\frac{1}{10}$ to that of local memory. Local memory used as disk cache is more difficult to scale than server cache when used in parallel applications simultaneously updating the same data because each update nullify the clients cache.

There is huge differences in how a computer access local data and how a computer serves multiple clients. For the local data the CPU cache provide a great boost. For a server with many clients there might be better to disable the CPU cache for data and instead use it for instructions. With many clients the randomization increases and the possibility for hit in the CPU cache decreases.

Extreme scalable file systems is possible when there is infrequent simultaneous change of the same data. AFS have proven to be scalable to more than 5000 machines.

Atomicity in file systems Atomicity have long been used in databases and have improved both ease of programming and performance. Databases have been highly optimized to handle parallel access. It is difficult to beat them in this game. Introducing atomicity to file systems might transfer this as well.

5.1 Reflection

Looking back at how the thesis were done there seem to be room for improvement. There were unexpected results, not to the science community but to me, extra work because of redesign, and tried but not used theories. The ideal walkthrough of this thesis would have been different.

Unexpected results When this thesis were started I believed that a latency reduction of 80% would increase performance. I also believed that copying data was straight forward and that larger blocks were better for performance. None of this is true.

Extra work When examining the PVFS code there first seemed to be one API for networked communications named *sockio*. The adaptation layer between the protocol and PVFS were designed and implemented for this. Then during tests I found out that this *sockio* API was only partially in use. This resulted in redesign and implementation as described in section4.4.1.

Implemented but removed As with the latency tests, alignment on the boundaries were assumed to increase performance. This were implemented and tested but found to not increase performance. For the sake of simplicity in the code these changes were left out in the final implementation but stored for possible further work.

To save traveling time I implemented a shared library with the same API as the SCI version so I could test things at home. The performance with the protocol and PVFS using this setup was very low because of my computer being a uniprocessor and the scheduler time slices mentioned in section 4.3.3.

The survey As mentioned above, clustering were new to me and some results were unexpected. One can say that I got more questions as I went along. I did a survey of different systems and technology to probe and find some of the answers. This survey have become larger than necessary, but at the time there were difficult to know what would become useful.

Collaboration My greatest regret for my work during this thesis was my individual stumbling. If I had asked the right questions the results would have been of more general interest.

Focus and pauses Some of the reason for this thesis were late was my spread interests. I had a pause for work and I tried to develop data structures for natural language and algorithms to query and manipulate this structure. In the end, most off all I just lost focus. The lesson is learned.

Chapter 6

Further work

The version of the SCI protocol that was developed during this thesis should be adjusted if used in production, the most notable is the way connections and disconnections are handled.

6.1 Changes to the SCI protocol

6.1.1 More dynamic connections and disconnection

The way connections and disconnections are done in the SCI protocol, all the nodes that a node may connect to or receive connections from have to be known at start time.

An alternative is to use TCP/IP for exchange of connection data. This should work very well since the connections are few and the performance impact is therefore negligible. Each node should set up a listening socket (TCP/IP) and connecting nodes should connect to this and exchange the SCI data needed to get the SCI communication running. Likewise at disconnection.

6.1.2 Token passing over TCP/IP

If the token passing system in section 4.3.4.4 prove beneficial, then moving the exchange operations to TCP/IP should be tested. Since TCP/IP have much more latency than SCI it should decrease performanc but with increased buffer sizes and holding times for tokens this might be nullified.

6.1.3 Buffer sizes

The size of the communication buffers have proven very essential for performance. Since the amount of memory that is sharable by SCI is finite then work should be done to tune the buffer sizes. Perhaps also dynamically as there may be other SCI applications running.

6.1.4 Test with higher performance SCI hardware

The SCI cards that was used had a throughput of 68-70MB/s. There is SCI cards available that have a throughput of 326MB/s. With these cards, the

performance of PVFS should be enhanced further.

6.1.5 Generic TCP/IP replacement using *LDPRELOAD*

If the SCI protocol should be used as a generic TCP/IP replacement, the wrapper functions should replace their socket equivalents in a copy of *libc* and be preloaded with *LDPRELOAD*. This should enable already compiled programs to use the SCI protocol as an TCP/IP replacement.

6.2 Other suggestions

6.2.1 Test PVFS with different file and record sizes

Chapter 2 show the impact of file and record size for local file systems. It would be interesting to do the same tests on PVFS.

6.2.2 Test scalability with PVFS

This is dependent upon getting the tokens exchange to work properly in PVFS and is crucial for the usability of PVFS with SCI.

6.2.3 NTP over SCI

System clocks in PCs drift and when updating them from another computer the latency of the network makes the new time slightly wrong. Because of this the accuracy of ordering debugging statements as described in section 4.4.2 is inaccurate.

A NTP daemon compensate for the drift by speeding up or slowing down the clock. To achieve further accuracy the NTP daemon, the NTP updates could be run over SCI for shorter latency. Even further accuracy could be achieved if the synchronizing computers first time the network latency between them.

6.2.4 Test with MPI-IO via ROMIO

Some of the reason PVFS was chosen for enhancement in this thesis was the support for MPI-IO via ROMIO. This should be no problem for the MPI-IO side alone but need integration with the existing MPI libraries for SCI. There is also the case that buffer sizes might need adjustment when dividing the shared memory between applications. Table 6.1 show how ROMIO provides the MPI-IO extention of MPI to applications using MPI while beeing layered on top of the modified PVFS presented in this thesis. Testing this should give the foundation for further tuning of performance.

6.2.5 Test latency impact on meta data operations

Meta data operations on PVFS is very slow. Decreasing latency might speed this up. If this proves successful the end-alignment on boundaries as described in section4.3.6 should be tried again. Since the implementation does writes in 128bytes sizes, performance might drop. This could then be masked for I/O (service number 7000) and only used for meta data (service number 3000).

<table>
<tr><td colspan="3">Application using MPI</td></tr>
<tr><td>MPI</td><td colspan="2">MPI-IO (ROMIO)</td></tr>
<tr><td rowspan="4">SCI</td><td colspan="2">PVFS</td></tr>
<tr><td colspan="2">Splitting and adaptation</td></tr>
<tr><td>TCP/IP</td><td>SCI protocol</td></tr>
<tr><td>Ethernet</td><td>SCI</td></tr>
</table>

Table 6.1: PVFS network layering

6.2.6 PVFS as a log-structured file system

The biggest advantage of PVFS too GFS is its possibility to cache huge amounts of data in the I/O servers memory. This both increase throughput and decrease latency. With dedicated I/O servers and maximum amount of memory in each the amount of cache hit should be excellent even if the data size is large. The log-structure will increase the write performance even further.

6.2.6.1 RAID striping to enhance data safety

As mentioned in section 2.4.4 the MTBF decrease with more components and since clusters consists of many components the MTBF might become uncomfortable low. Doing the RAID parity logic on the log segments will remove the read-modify-write sequence that strain write performance.

This have earlier been tried in Zebra and xfs with good results according to the papers published on the subject.

Bibliography

[ADN+95] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless network file systems. In *In Proceedings of the 15th Symposium on Operating System Principles. ACM*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[Aru96] Meenakshi Arunachalam. Implementation and evaluation of prefetching in the intel paragon parallel file system, 1996.

[Bae02] Troy Baer. Parallel i/o experiences on an sgi 750 cluster. Web at http://www.osc.edu/~troy/cug_2002/, 2002.

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[BHK+91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212. Association for Computing Machinery SIGOPS, 1991.

[CD91] Luis-Felipe Cabrera and Darrell D.E. Swift: Using distributed disk striping to provide high i/o data rates. *Computing Systems*, 4(4):405–436, Fall 1991. Earlier than Zebra.

[CLRT00] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[Cor97] Toni Cortes. *Cooperative Caching and Prefetching in Parallel/Distributed File Systems*. PhD thesis, Barcelona, Spain, 1997.

[DAP88] Randy H. Katz David A. Patterson, Garth Gibson. A case for redundant arrays of inexpensive disks (raid). In *International Conference on Management of Data (SIGMOD)*, pages 109–116, 1988. The first published Berkeley paper on RAIDs, it gives all the RAID nomenclature.

[DMST95] Murth Devarakonda, Ajay Mohindra, Jill Simoneaux, and William H. Tetzlaff. Evaluation of design alternatives for a cluster file system. In *USENIX 1995 Technical Conference Proceedings*, January 1995.

[GLT99] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[GNS88] David K. Gifford, Roger M. Needham, and Michael D. Schroeder. The cedar file system. In *Communications of the ACM*, volume 31. Association for Computing Machinery, March 1988.

[Gon01] Li Gong. Project jxta: A technology overview. White paper on web at http://www.jxta.org/project/www/white_papers.html, April 2001.

[HK93] John H.Hartman and John K.Ousterhout. The zebra striped network file system. In *Proceedings of the fourteenth ACM symposium on Operating systems principles.*, pages 29–43. Association for Computing Machinery, Association for Computing Machinery, December 1993.

[HLM94] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the Winter 1994 USENIX Conference*, pages 235–246, San Franscisco, CA, January 1994. USENIX.

[HS96] R. L. Haskin and F. B. Schmuck. The Tiger Shark file system. In *Proceedings of the 41st IEEE Computer Society International Conference (COMPCON '96)*, pages 226–231, Santa Clara, CA, USA, 25–28 1996.

[Inf00] Infiniband architecture: Next-generation server i/o. White Paper at http://www.dell.com/us/en/arm/topics/vectors_2000-infiniband.htm, 2000.

[Kle86] S. R. Kleiman. Vnodes: An architecture for multiple file system types in sun unix. In *USENIX Conference Procedings*, pages 238–47, Atlanta (GA), Summer 1986. USENIX.

[mpi97] Message Passing Interface Forum. MPI-2: Extensions ot the Message-Passing Interface. Web at http://www.mpi-forum.org/docs/docs.html, July 1997.

[MRZ02] Joshua MacDonald, Hans Reiser, and Alex Zarochentcev. Reiser4 transaction design document. Technical report, The Naming System Venture, January 2002.

[NWO88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[pos] *IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)-part 1: System program interface (API)[C language], 1996 edition.*

[Ras94] R. Rashid. Microsoft's tiger media server. In *The First Networks of Workstations Workshop Record*, October 1994.

[RO91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 1–15. Association for Computing Machinery, October 1991.

[RS81] D.P. Reed and L. Svobodava. Swallow: A distributed data storage system for a local network. In *Local Networks for Computer Communications*, pages 355–373, 1981.

[RT74] Dennis. M. Ritchie and Ken Thompson. The unix time-sharing system. In *Communications of the ACM*, pages 365–375, July 1974.

[Sat90] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, pages 9–20, May 1990.

[SCI91] Sci-scalable coherent interface, November 1991. Draft for Recirculation to the Balloting Body.

[SCI93] IEEE Standard for Scalable Coherent Interface (SCI), August 1993.

[SDWH+96] Adam Sweeney, Doug Doucette, Curtis Anderson Wei Hu, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *USENIX 1996 Annual Technical Conference*. The USENIX Association., January 1996.

[SEP+97] S. Soltis, G. Erickson, K. Preslan, M. O'Keefe, and T. Ruwart. The global file system: A file system for shared disk storage. Web at http://citeseer.nj.nec.com/soltis97global.html, 1997.

[SGK+85] R. Sandberg, D. GoldBerg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *In Proceedings of the Summer 1985*, pages 119, 130. USENIX, June 1985.

[SKK+90] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[Sol97] S. Soltis. *The Design and Implementation of a Distributed File System Based on Shared Network Storage.* PhD thesis, Ph.D. Dissertation, Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, August 1997.

[WBvE97] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Incorporating memory management into user-level network interfaces. Technical Report TR97-1620, 13, 1997.

Appendix A

Vocabulary

ACID Atomicity, Concurrency, Isolation and Durability. Is used mostly about database environments as they implements the ACID characteristics more or less.

AFS is a distributed file system product. It was a part of the Andrew project pioneered at Carnegie Mellon University and this was the Andrew File System (AFS), but it have since then evolved beyond the Andrew project and the acronym no longer yields.

bandwidth is the maximum throughput.

bottleneck a part of a system that when increased performance, the overall performance is increased.

buffering is the temporarily storage between communicating components used in one direction.

cache is the temporarily storage used between communicating components where one component is slower than the other. The fastest component can retrieve some of the wanted data from the cache by storing data sent to the other component also in the cache.

CFS Cedar File System.

cluster The Widest Definition: Any number of computers communicating at any distance. The Common Definition: A relatively small number of computers communicating at a relatively small distance (within the same room) and used as a single, shared computing resource.

CODA Constant Data Availability is a distributed file system based on the Andrew File System. It supports disconnected operation and server replication.

CSMA/CD Carrier Sense Multiple Access Collision Detection.

CVS Concurrent Versions System is a version control system. It is used to record the history of source files.

DBMS Data Base Management System.

DFS Distributed File System. It is a distributed implementation of the classical time sharing model [RT74] of a file system, where multiple users share files and storage resources. It is a file system, whose clients, servers , and storage devices are dispersed among the machines of a distributed system. Accordingly, services activity has to be carried out across the network, and instead of a single centralized data repository there are multiple and independent storage devices.

DMA Direct Memory Access.

Ext the Extended File system was the first file system designed specifically for Linux and was introduced in April 1992.

Ext2 the Second Extended File system was an improvement to

EXT and was added to the Linux kernel in 1993.

Fault tolerance The ability of a system to respond gracefully to an unexpected hardware or software failure. There are many levels of fault tolerance, the lowest being the ability to continue operation in the event of a power failure. Many fault-tolerant computer systems mirror all operations – that is, every operation is performed on two or more duplicate systems, so if one fails the other can take over. *Wĕbopēdia.*

FS File System A file system provides file services to clients. A client interface for a file service is formed by a set of *file operations.* The most primitive operations are Create a file, Delete a file, Read from a file, and Write to a file.

GFS Global File System.

GPL General Public Licence.

HIPS Highly Integrated Parallel System

HPC High Performance Computing is systems tuned for performance in highly integrated parallel systems.

HTC High Throughput Computing is systems tuned for throughput and computational power. Large systems like SETI at home is a High Throughput Computing (HTC) with High Performance Computing (HPC) nodes. It is more loosely coupled than HPC.

I/O-path the travel path for data from source to destination.

i-list provides the operating system with a map into the memory of some physical storage device. The map is continually being revised, as the files are created and removed, and as they shrink and grow in size. Thus, the mechanism of mapping must be very flexible to accommodate drastic changes in the number and size of files. The i-list is stored in a known location, on the same memory storage device that it maps.

i-node is an entry in an i-list who contain the information necessary to get information from the storage device, which typically communicates in fixed-size disk blocks.

JBOD Just a Bunch Of Disks is a collection of disks without any higher level handling.

LCPS Loosely Coupled Parallel System

LFS Log-structured File System.

libc is the C standard library.

MPI Message Passing Interface.

MPI-IO The I/O chapter of MPI.

MTBF Mean Time Between Failure.

NFS Network File System is described in [SGK+85], was developed by Sun and is now one of the most used distributed file systems around.

NOW Network Of cooperating Workstations.

NV RAM Non Volatile RAM.

N-WAL Neighbor Write-Ahead Logging uses neighbors to store the log while in memory while waiting for disk. The log is duplicated across at least two neighbors.

OpenAFS IBM branched the source of the AFS product, and made a copy of the source available for community development and maintenance. They called the release OpenAFS. It's available from the IBM Open Source site.

PCI Peripheral Component Interconnect is a local bus standard developed by Intel Corporation.

PVFS Parallel Virtual File System.

PVM Parallel Virtual Machine.

RAID Redundant Array of Independent Disks.

ROMIO a portable implementation of MPI-IO, the I/O chapter in MPI-2. For more, visit http://www-unix.mcs.anl.gov/romio.

RT-DBMS Real Time DBMS.

SCI Scalable Coherent Interface was introduced as a draft [SCI91] in 1991 and became an IEEE standard [SCI93] in 1993. It is a hardware architecture for shared memory. Main benefits are low latency and high throughput. This does it suitable for clustering environments.

throughput is different from bandwidth in the way that bandwidth is the maximum throughput.

***nix** Unix compliant platform, this include both systems that rightfully call them self Unix and systems that are just compliant.

vnode virtual node The vnode interface was invented over a decade ago to facilitate the implementation of multiple file systems in one operating system [Kle86], and it has been very successful at that. It is now universally present in Unix operating systems.

WAFL Write Anywhere File Layout.

warm / cold cache Warm cache means that the cache have been filled with so much data that it might be after run for a long time. Cold cache means that there is no data in the cache.

write-invalidate cache protocol where write operations invalidates other cached copies of the same data.

Index